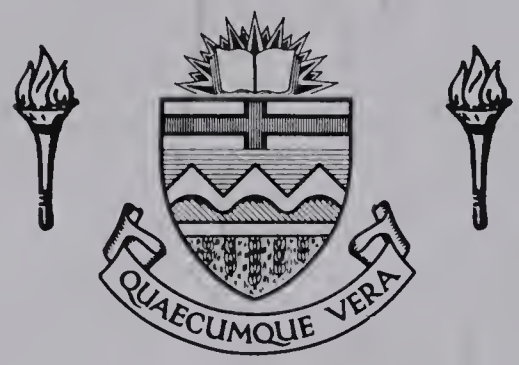

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Mohammed Abdul Waheed

TITLE OF THESIS On the Design of Directly Executable Representations

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED FALL, 1981

THE UNIVERSITY OF ALBERTA

On the Design of Directly Executable Representations

by

© Mohammed Abdul Waheed

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

FALL, 1981

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled On the Design of Directly Executable Representations submitted by Mohammed Abdul Waheed in partial fulfilment of the requirements for the degree of Master of Science.

Acknowledgements

I would like to thank Professor W. S. Adams for all his encouragement and advice. I have appreciated the useful comments and criticism on content and style made by my committee, Professors S. Dasgupta, K. W. Smillie and S. J. Titus. Special gratitude to Professor A. Brindle who helped shape the original draft.

Mat, MM, QB and Sam offered substantial moral support. Finally, thanks go to Atul, Chris and Kha Sin for churning out the final release.

Abstract

The development of high level languages, which eased the problem of program writing, has been done independently of advances in machine design. Machines have been designed to suit particular languages and languages designed for particular machines, but these are too restrictive and the semantic gap is still increasing.

It has been shown that the conventional method of executing programs by compiling them to host instructions or interpreting the source directly is not efficient. A more efficient method is to execute an intermediate form on a 'soft architecture'.

Several intermediate forms have been developed and shown to be efficient. Notable among them is the directly executable version of FORTRAN (DELTRAN) developed at the Stanford Emulation Laboratory, which takes about five times less space and time than a machine language representation.

This thesis surveys language–architecture incompatibility, determines the analytical conditions under which interpretation is efficient, surveys the directly executable languages developed and suggests design approaches for better intermediate representations.

Table of Contents

List of Figures

Dedicated to Mummy and Phool who made me what I am today.

1. Language - Architecture Incompatibility

1.1 Introduction

There are innumerable ways of expressing users' algorithms and executing them. "High level" languages have been and are still being designed to make it easier to write programs. Machines are also being designed with a bias towards particular languages but the so called "semantic gap" between languages and machines continues to exist [FLYN74],[MYER78].

The wider the gap, the less likely it is that the user will be able to write programs which minimize the time and space needed in a real machine to execute his algorithms. The problem of bridging the gap is, of course, the domain of translation programs such as compilers or interpreters or some combination of these. Before a source language program is completely executed it may have to undergo a sequence of intermediate transformation phases. The user is seldom aware of these intermediate program representations as they are not usually made accessible nor are they intended to be understood or modified by users. These internal representations can be regarded as programs for abstract machines with characteristics intermediate between high level language machines and real "machine language" hosts.

The concept of intermediate machines was first explored by Iliffe [ILIF68] and has attracted interest since then as a theoretical and practical tool for improving the efficiency of executing high level language programs. By choosing the optimal "distance" between the two extremes it may be possible to improve the time and space needed to meet the user's needs. Hoevel [HOEV74] presented an analytical argument showing that executing an intermediate form of a source language program would be faster than executing the source itself or executing the equivalent program in the host language. Studies done at Stanford Emulation Laboratory showed that a directly executable version of Fortran (DELTRAN) executed five times faster and took five times less space than its machine language equivalent on an IBM 360 computer [FLYN77]. A few other intermediate languages for direct execution have been developed and shown to be efficient.

In addition to executing intermediate languages, which is the topic of this thesis, there have been other approaches to executing programs conveniently and efficiently.

It is also important to have compact representations of the programs. This not only saves space but also fetch time when this appears to be a major factor. On the other hand, other factors like the complexity of decoding may offset the gains due to compacting the code. A few such intermediate languages for direct execution have been developed and shown to be efficient.

This thesis surveys the mismatch between program representation and the architectures of the currently popular machines, determines conditions for efficient interpreting, surveys the directly executable and interpretable languages developed, and suggests design approaches for better interpretable representations.

Chapter One discusses historical changes in architecture and programming styles. Results of experiments analyzing architectures are reviewed as are different methods of executing programs, high level language machines, forms and applications of intermediate languages and universal host machines and tagged architectures.

In Chapter Two a review of analytical conditions to determine conditions under which executing programs by interpreting is efficient, and conditions under which translating sections of source program into micro code and storing them in control store speeds program execution, is made. Based on this analysis other models are developed and conditions for efficient interpretation determined.

Chapter Three discusses the need for and characteristics of directly interpretable representations. Existing DIR's are reviewed, and design approaches for better DIRs are presented.

1.2 The Problem

The earliest calculators were designed to do simple arithmetic. Computers evolving from these early calculators were best programmed in their respective machine languages. With advancements in technology the underlying hardware of computers changed from mechanical to electronic, moving from vacuum tubes to transistors and finally to integrated circuits. The tremendous increase in speed thus achieved made it possible to run large programs quickly, but it was difficult to write such programs in machine language. High level programming languages and their corresponding translators made it possible to program the machines conveniently.

The earliest approach was the UNCOL/POL approach started in 1958, in which Strong *et al.* [STRG58] tried to design a universal intermediate language into which any high level language program could be reduced and which could be compiled for any machine. The motivating factor was the possibility of solving the problems of portability and having to write separate complete compilers for each new machine and language. The idea was to divide the job of compiler writing into two parts and to avoid writing the second half for each new machine. The project did not meet with much success and was abandoned.

To overcome the inefficiencies of compilers languages were designed for existing architectures and machines with special features designed to suit the specific high level language (notably the Burroughs B5500 for ALGOL). Languages are being designed to suit particular architectures, and machines designed on the basis of statistics of program execution.

These methods of designing machines and languages are too restrictive. Language designers seem to expect an undue amount of flexibility in order that users' algorithms can be easily and efficiently coded into them. Also system managers would like to have the choice of running many different languages on their machines. Languages designed for particular architectures are not user efficient, and machines designed by collecting statistics from a particular environment cannot be optimal for all environments. Though the studies did not result in efficient execution of programs, they have given rise to promising approaches in machine design.

Another approach is to design machines for individual languages, i.e. to have the high level language as the machine language. The few such machines, such as DPIL, ADAM, SNOBOL4 [CHUY75], that have been built are mini computers and can be used for programs written in that particular language only. The single execution phase is very complex in all of these machines.

A more general approach is to find a better means of executing programs with existing languages and architectures and then suggest improvements for future designs. This approach will potentially reduce the overhead caused by instructions that sequence through the program and the instructions that move data around in slower devices like main store.

The capabilities of computers were also increased by features such as large memories and several input and output devices. This led to using machines more for data manipulation rather than just for simple arithmetic. To make such extensive use possible, modern software, such as operating systems, has been developed to automate console control and manage resources.

Attempts have been made to define the terms *machine (computer)* and *computer architecture*. A broad definition by Flynn [FLYN80] is as follows:

An *instruction* is a function with a set of states as its range and as its domain. Given a particular state it causes a state transition, giving the next state. A *machine* is a function plus a mechanism that causes state transition. Its range is a set of instructions and its domain the storage. An *emulator* is a program that defines the function of a machine. An *interpreter* is a program that interprets an instruction into micro code. All the user accessible aspects of the instruction set of the machine form the *architecture*. Architecture is independent of hardware implementation and technology.

A *high level language* is a language that allows the programmer to use symbolic operators to signify operations and symbolic names to represent data and data structures and has syntax and semantics to describe the algorithm conveniently. Examples of high level languages are FORTRAN, COBOL, PASCAL, APL and LISP.

A *translator* is a program that translates a high level language program into an intermediate language which is distinct from the machine language. A *compiler* is a program that translates a high level language program into its machine language or microcode equivalent.

With the increase in variety of applications of computers, the number and types of high level languages have also been increasing. These different languages need different architectures for their efficient execution.

Software has developed rapidly and in many different directions, while the logical structure of machines or architecture is almost unchanged from von Neumann's design. Because of this the concepts in high level programming languages and the concepts in architectures are now incompatible. This thesis examines several methods of bridging the gap between high level languages and architectures.

A user's problem has to go through many phases before it can be 'understood' by the machine. These phases are broadly illustrated by Figure 1.1.

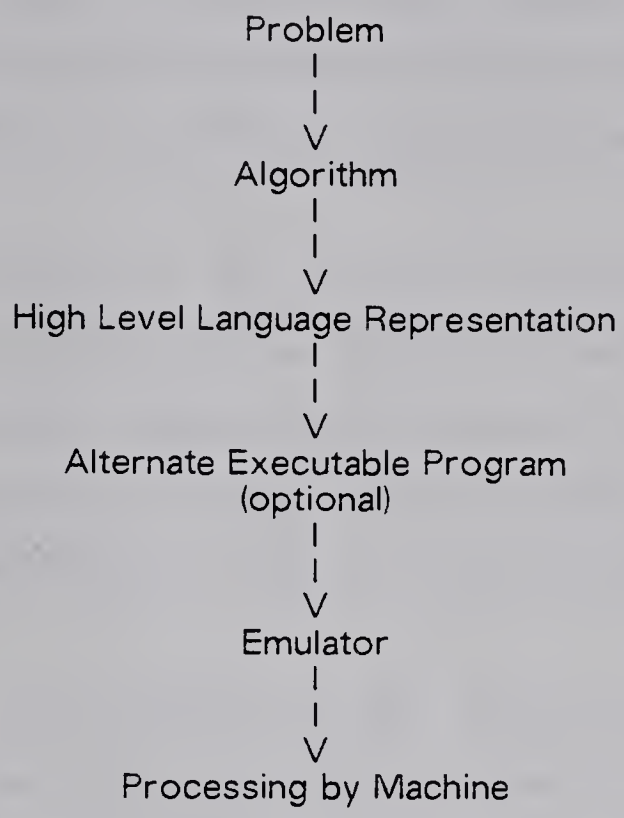

Figure 1.1. Transformation of a problem to machine processing.

The problem to be solved should be clear and well defined. It can at most be modified to make it representable in the language (finding an appropriate data structure for the problem, for example). An efficient algorithm for the problem should be found. An inefficiency in the algorithm is difficult to compensate for in future steps. Choosing an efficient algorithm is an important and difficult step, but since it is not directly related to this thesis it will not be discussed.

A programming language suitable for the algorithm should be selected. Trying to solve a recursive problem using FORTRAN, or scientific computation with LISP are examples of what should not be done.

Coding the algorithm into a programming language, especially a block structured language, is also an important step. Variables used in a particular block can be declared

locally in the block or globally with the program enclosing the block. Both work the same way for a programmer but the former is more efficient when loading variables along with blocks. If the number of registers available is less than the number of variables, the variables have to be moved back and forth. If only necessary variables are given in a required block, a better 'program working set' will be produced. Though the above factors play a significant role in efficient program representation, they will not be discussed in this thesis.

Efficiency here is defined in terms of time to execute the executable program and space to store the executable program. The time for execution includes the time to translate or compile the source program, time to interpret the executable representation, plus time to execute the ultimate microcode. The space required is the space to store the executable program representation. It does not include space to store the source program.

The creation of the executable program representation from the high level language program is one of the most important steps, because the earlier the inefficiencies are introduced the more they propagate through the later steps. One method is compiling the high level language program to the machine's instruction set. Another method is to tune[1] the machine to accept the language directly. As will be shown later these extreme methods are inefficient.

Another approach is to translate the language to an intermediate form which is accepted by the emulator. The level of an optimal intermediate form is an open problem; the intermediate form should, however, aim at eliminating inefficiencies in both the intermediate form and its execution.

The format and size of a directly executable intermediate form is subject to many tradeoffs. The source itself would be a compact form of representation but its direct execution by the emulator would require complex decoding. On the other hand the microcode version of the source language program would be the fastest to execute but would make the program's representation very large, which could more than offset the execution time saved, by increasing the fetch time.

---

[1]tuning here refers to a user microprogramable architecture eg. B1700/B1800 in which the instruction set for the programmer is defined by microcode.

Finally, there is very little a user can do about the execution phase. The machine can be modified by changing the contents of its control store. The advent of writable control store has made it possible to modify architectures for efficient representation of source programs. User microprogrammable machines allow users to modify the architecture to suit their requirements. This is as far as the user can go in modifying the architecture to adapt the language. This phase of program execution will also be considered as fixed and not discussed in detail.

1.3 The Divergence of Architectures and Programs

Since the advent of transistor technology in the 1950's developments in computer technology have been remarkable. By the 1960's considerable advances had been made in the field of computer science, besides those in architecture. Machine design had been the concern of engineers and technologists who gave more emphasis to technology and very little to users needs.

The uses of machines have been simultaneously and independently growing. The many high level languages developed for different kinds of problems have required different architectures for their efficient execution. In the absence of such architectures the software required to implement languages has been expensive and has required a lot of memory. Observations on these points are now leading computer scientists to question the effectiveness of von Neumann architectures.

Since von Neumann developed the stored program concept most machines have been organized around a single sequential memory. Sequential architecture was efficient when program and data structures were all sequential (such as FORTRAN using one dimensional arrays), but the introduction of recursion to languages and multiple dimensions to data types now requires many instructions to access data. This makes von Neumann machines inefficient in applications employing these features.

Thus, in the last decade or so it has been observed by several computer scientists that machine architectures are diverging from what programs and programming languages require for their efficient execution.

"It is accidental that digital computers are organized such as desk calculators. With some worse luck we might have taken Turing machine as our model. And someone would have been unenlightened enough to prove that, under certain (actually untrue) assumptions, it made no difference." ..... "If design engineers began to seek more efficient encodings for commonly used sequences of instructions (for instance, direct addressing instead of a sequence of tape moving commands), progress towards the modern computer would have begun." [MCKE67]

"Most present day computers have an architecture designed in the early 60's. They have remained substantially unchanged for a decade in the name of compatiblity in spite of their obstacles to generating efficient code from high level languages"..... "The fact that few compilers for third generation computers can produce code that even comes close to what a skilled assembly language programmer can generate argues strongly for redesigning machine architectures so that compilers can do their jobs better." [TANE78]

"there have been no advances in the computer architectures of current systems since the 1950's. Some so called advances that might come to mind(eg. cache memories,instruction pipelining, the microprogramming concept) are not computer architecture advances; they are processor architecture or organization advances. In fact, some of these implementation advances can be viewed as step backward in terms of computer architecure." [MYER78]

"Not all architectural innovations have been of benefit to the programmer. In many cases there has been little concern for how the new features would affect the ease with which correct software may be prepared. In fact, as we shall see, many recent advances in computer architecture have presented the programmer with new and difficult challenges rather than making his task easier. The introduction of large, baroque instruction sets -- while presented as aids to programmers -- may have had an overall negative software impact. The use of high level languages may be inhibited because it is difficult to utilize 'special architectural features' from within a standard language."[DENN79]

"It should not surprise the reader that traditional image machines such as the IBM system 360, the UNIVAC 1100, and the DEC-10 are not optimal program representations. They were conceived at least a decade ago (and in some cases their designs may be traced back to almost three decades) in contexts that are simply no longer valid." [FLYN80]

One possible reason for lack of advances in the architecture is that most languages were designed in academic environments, whereas the machines were built by commercial organizations.

1.4 Some Architectures And Their Deficiencies

According to Myers [MYER78] all architectural concepts were designed and implemented in the 1940's and modern designs are just faster. Fixed architectures and changes in languages and problems have created a "semantic gap".

The following are a few reasons for this gap. Data structural concepts like arrays and records were developed in programming languages, but no corresponding development appears in the architectures. Similarly architectural features to facilitate string processing, procedures, block structure, and data representations are lacking. Most of these features are implemented solely through software, such as compilers, at the cost of increasing complexity of these representations thus making executable representations inefficient.

The following studies indicate that present day architectures are not efficient for representing a user's program. The decreasing price of hardware may help machine designers in the future, but for the moment, work needs to be done on finding better forms to represent programs.

Very little interest has been shown in analyzing architectures with respect to data movement, register usage, memory addressing and the overhead in extra instructions needed to execute a user's instruction. Several reasons can be cited for this indifference. The amount of work involved is excessive and it serves little purpose in influencing industries. Many machines have to mimic popular architectures (such as IBM's) to be able to use their software.

The amount of memory, registers and cache associated with machines has, on the most part, been arbitrary. The rise in computing speed and decrease in hardware costs are possibly reasons for manufacturers to provide increased amounts of register space, memory and cache without justifying the amounts. Actual measurement of architectures, if done by the manufacturers, has not been reported.

A few studies report on register usage during program execution on different architectures. Notable among them are the studies of Flynn's group, which analyzed data movements, and of Lunde, who determined the optimal number of registers for DEC-10. Details of these and other studies are given below.

1.4.1 Functional and Non-functional Instructions

A study comparing the architectural constraints of the IBM 7090 and IBM 360 was reported by Flynn in 1974 [FLYN74]. In this study, Flynn computed the ratio of functional to non-functional (overhead) instructions. He classified the various kinds of instructions as follows:

M-Type instructions are instructions that move data around without doing any actual computation on them (e.g. LOAD, STORE, MOVE).

P-Type instructions are instructions that sequence through the program without performing any tranformations on data (e.g. BRANCH, GOTO).

F-Type instructions are functional instructions that perform computations on the data (e.g. ADD, SUBTRACT, AND, OR).

Flynn defined M-type and P-type instructions as non-functional.

Analysis was performed on the data collected by Winder [WIND73], and the results indicate that 7090 is not inferior to that of the 360 as expected. The IBM 7090 was a machine with many limitations (it had only one accumulator, for example) but the ratio of functional to non-functional instructions does not support it.

IBM 7090

M-Type/ F-type = 1.96

Accumulator based M-type/Floating point F -type = 1.24

P-Type/ F-type = .81

P-Type/ Floating point F-type = 1.67

IBM 360

M-Type/ F-type = 2.9

General Purpose Register based M-type/General Purpose based F - type = 3.1

P-Type/ F-type = 2.5

P-Type/ Floating point F-type = 4.5

or summarizing

IBM 7090	IBM 360
(P + M)/F = 2.8	*(P + M)/F* = 5.5

The results do not imply that the 360's architecture is worse than the 7090's. The ratios

just indicate poor register usage, which could possibly be due to inefficient register allocation by compilers. Of the sixteen registers in the 360, between three and six are used for addressability and linkages, leaving the user's program with less than what might be normally required and thus sometimes forcing unnecessary data movement. Also the single accumulator 7090's architecture cannot be declared superior to the 360, because it has a limitation on the size of memory it can address. The 360's registers no doubt increased this limit.

Lunde in 1975 [LUND75] found the ratio of functional to non-functional instructions on the DEC-10 to be:

$$M/F = 1.5 \qquad P/F = 1.1 \qquad (M + P)/F = 2.6$$

These ratios are better than those found for the IBM 7090, the IBM 360 and the PDP-11 (quoted below).

Neuhauser[NEUH80] made a detailed study analysis of the PDP-11's architecture and found the ratio of the functional to non-functional instructions to be:

$$M/F = 2.6 \qquad P/F = 3.7 \qquad (P + M)/F = 6.3$$

These ratios are worse than that of the IBM 360, IBM 7090 or DEC-10's.

Tafvelen *et al.* [TAFV75] emulated a stack-oriented machine with virtual memory and 0, 1, or 2 address fields, suitable for their directly executed language DEL-Mary. Their machine gives the following ratios:

$$M/F = 1.69 \qquad P/F = 0.87$$

The reason for the similarity of this M/F ratio is, according to them, that a certain amount of data movement cannot be avoided. However this similarity could possibly be due to the stack architecture.

Though it is difficult to judge which architecture is best, these figures suggest that there have been marked changes in programming styles and that compilers may be inefficient in allocating registers.

Measuring architectures is difficult and has not been done rigorously. Flynn's distinction between functional and non-functional instructions is not valid for all programs. Sorting is accomplished by almost all non-functional instructions. Not all ADD instructions are functional. All testings are not overhead instructions (eg. in searching). A better measurement would have been to count the functional and non-functional

instructions in the source and the representation to compute the ratios.

1.4.2 Optimal Numbers of Registers

The term ISP (instruction set processor) coined by Bell and Newell [BELL71] is defined as the logical processor that processes the instruction set. Different ISP's have different word lengths, numbers of instructions, general and index registers. An attempt to determine an optimal ISP was first made by Lunde [LUND77]. Experiments were performed to test the ISP of the DEC-10. These consisted of analyzing register life. Register life is the time between 1) the load and 2) the register's last use before loading again. He used a set of programs and recorded information for every instruction executed. The test data were forty one programs coded in four languages (ALGOL, BASIC, BLISS and FORTRAN) by four different programmers.

A two phased process was used to determine the length of time for which the register was not active between two consecutive loads. Phase one detected register life and classified it according to the activity and noted the descriptions into a file. Phase two worked on the data collected by phase one and determined the number of registers live at each time and other statistics. The results obtained were as follows:

About 68% of the lives(averaged over all data) are between 2 to 7 instructions long, 4% more than 32 instructions long, and about 20% between 8 and 31. The lifelength for chosen individual programs varied between 4 and 24 instructions with an average of about 12 instructions. Only between 2 and 6 registers were simultaneously active. According to his final conclusions, which are biased towards the programmer, language and the compiler used, eight registers would be sufficient for the general register ISP similar to DEC-10, provided they are not used for addressing or indexing. Though this does not mean that eight is an optimal set of registers for any machine, such techniques can be used to evaluate lifetimes before deciding on the number of registers. Lunde's results are not valid when the same registers are used for addressing too. But they definitely are inexpensive methods that can give meaningful results. Similar studies to measure the size of control store required, and the cache necessary to buffer information have not been determined. Techniques to save space and time have been developed independent of the architectures. Measuring memory accessing techniques is

also required.

1.4.3 Information Content of Addresses

Hammerstorm in 1977 [HAMM77] studied memory systems and CPU memory-addressing architectures by developing models. He applied information theory to an 'R + D' architecture; such machines compute a memory address by taking the contents of a memory register R and adding a displacement D. This type of addressing is the most commonly used. Using different addressing schemes, based on the IBM 360's formats, he found that the information content in a sequence of address computations is less than 2% and the remaining 98% is predictable. His paper suggests techniques for better methods of addressing memory by 'tuning architectures'.

1.4.4 EM-1's Architecture

Tanenbaum [TANE78] designed an architecture on the basis of the analysis of 10,000 lines of programs coded in the structured high level language EM-1. His main idea was to have a compact representation of the source language program. The design permits efficient implementation of most frequently used statements, and uses a small but fixed opcode and address field for the ones frequently used. The architecture helps minimize program size and makes computation straightforward. His architecture, on a sample test of four programs, gave average compaction ratios of 2.2 and 3.45 on the size of code (in bits) produced for the PDP-11 and CDC-CYBER respectively.

1.4.5 Analysis of the PDP-11's Architecture

Neuhauser [NEUH80] made a detailed study of the PDP-11's architecture by monitoring program execution and collecting statistics. He found the PDP-11's instruction set to be inefficient. This, according to him, is possibly due to orthogonality. In architecture the level of orthogonality refers to the number of overlapping concepts. A lower level of orthogonality implies a large number of overlapping concepts such as many different ways to decrement a register. In the PDP-11 orthogonality is expressed in two forms: in combining arbitrary operators and operand address modes and in pairing operand data types and operators. This orthogonality increased the required instruction

length. Neuhauser observed that a small subset of the instruction set is used most of the time. Simple operators dominate the processor whereas the complex ones are rarely used. DELTRAN (the directly executable version of FORTRAN) has been found to be a better executable representation than machine language instructions for the PDP-11's architecture. He concludes that "substantial improvements can be made by structuring processors for dedicated use in the high level language environment."

1.5 Different Methods Of Executing Programs

Including the conventional method of executing programs by compiling the source program to host language instructions, the methods used to execute programs can be represented as:

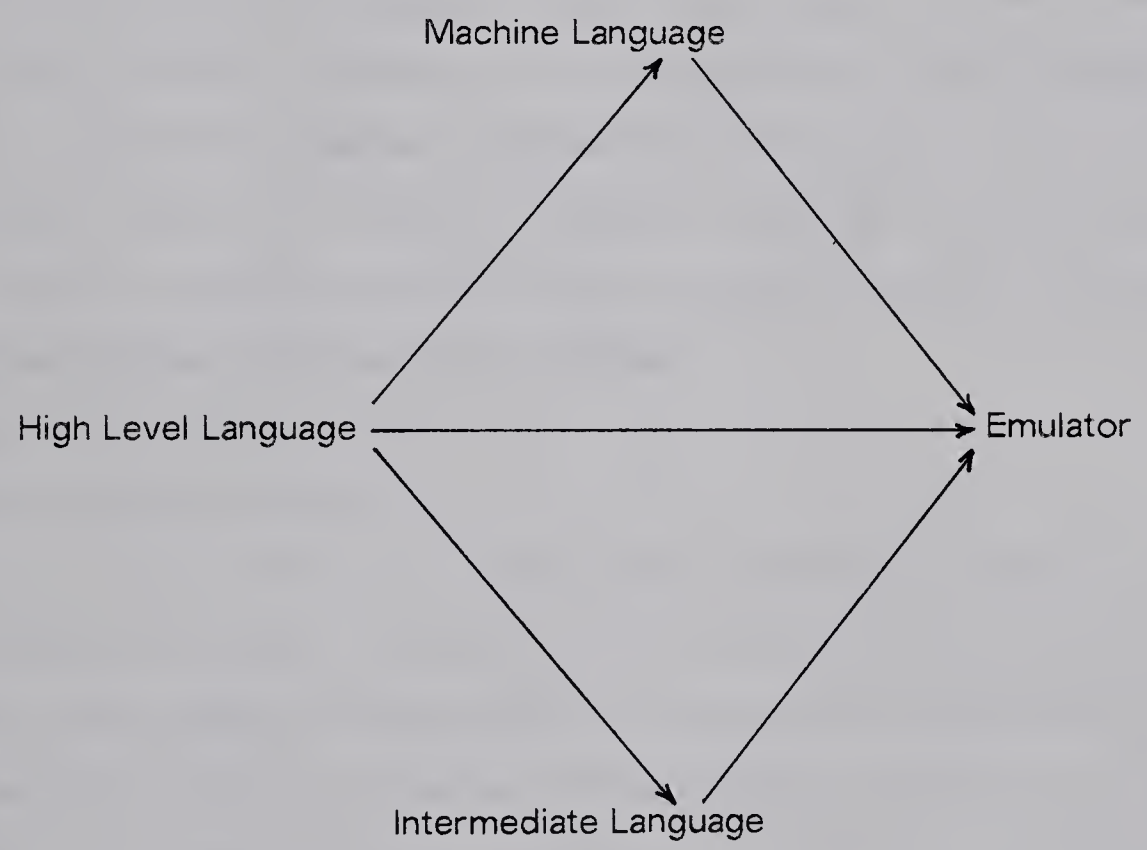

Figure 1.2. Program execution methods.

Work on the methods of execution represented above (excluding the conventional method of compilation) can be categorized as:

1. High Level Language machines

2. Intermediate Languages
   a. Machine–dependent intermediate language
   b. Language–dependent intermediate language
3. Universal Host Machines and Tagged architectures.

1.6 High Level Language Machines

High Level Language ———————> Host
(with no intermediate stage)

In order to avoid compilers and simplify operating systems, some high level language machines have been designed. A high level language machine is defined as a machine whose emulator is capable of processing the high level language program directly, i.e., without any compilation, interpretation or creation of a new program form.

In these machines the high level language is also the machine language of the machine. It has been claimed that such machines are easy for the user to understand. Chu [CHUY75] has put these machines into two classes:

1. Indirect execution architecture
2. Direct execution architecture

From the user's point of view both of these types execute the program in a high level language directly without any compilation. It is the instruction set of the machine that distinguishes them. Indirect machines interpret the source with the hardware and execute it. This execution technique is the same as implementing an interpreter in the hardware. An example of an indirect execution architecture is the SYMBOL machine announced by Rice *et al.* in 1971.

Direct execution architectures execute the high level language statements directly with the hardware. Machines of both kinds have been designed and built. Chu *et al.* have been working on on direct execution architectures since 1967.

The first step towards a high level language computer architecture was the Burroughs B5000, designed in 1961, and later improved to the B5500 and then to the B6000/6500. These machines were not designed to execute a high level language

directly but had most of the features necessary to execute a high level language, such as stacks, implemented in the hardware.

Very few high level language machines have actually been built. Notable among them are:

ALGOL processor: Anderson in 1961 [ANDE61] designed a processor for direct execution of ALGOL 60 programs. The architecture is an extension to the B5500 and has three stacks that control states, arithmetic operators and operands.

FORTRAN processors: Melbourne and Pugmire in 1965 [MELB65] proposed a microprogrammed machine for FORTRAN. Their machine, which could be used for small FORTRAN programs, evaluated arithmetic expressions by converting them to reverse polish form. Another FORTRAN machine which did most of the interpreting with the hardware was designed by Bashkow[CHUY75].

EULER processor: Wirth and Weber in 1967 [WEBE67] developed a microprogrammed processor for the high level language EULER. The processor was implemented on an IBM 360 and programs were executed indirectly. It had a microprogrammed translator that translated the arithmetic expressions in EULER programs into reverse polish strings and a microprogrammed interpreter to interpret it, and a 360 machine language program that controlled the two.

PL/1 processor: Sugimoto in 1961 [SUGI69] proposed a processor for PL/1. His processor comprised a reducer to translate the source and a hardware direct processor to interpret it.

None of the designs have been implemented. Only simulations were conducted and it is suspected that some of the techniques used would not allow large programs to be executed on them efficiently.

ADAM processor: Mullery *et al.* in 1964 [MULL64] designed a processor to directly execute a language also designed by them. The important feature of this processor is that it could handle variable length data.

SYMBOL processor: Rice and Smith in 1971 [RICE71] announced a high level language machine SYMBOL. SYMBOL is one of the first machines actually built. The machine processes programs written in the high level language SYMBOL, which is similar to ALGOL, PL/1 and EULER. The source program is translated to reverse polish strings

which were executed by a central processor (CP). Expressions are evaluated through stacks where both the operands and results are stored.

B1700: One of the problems with the SYMBOL machine was its restriction to the one language SYMBOL. This was overcome by Wilner in the B1700 [WILN72]. This machine placed the language's interpreter in the control store, thus tuning the machine to the language being executed. The B1700 also executed programs through intermediate languages.

APL processor: Hassitt [ZAKS71] and Abrams[CHUY75] were among the many people that developed APL machines. APL, as a mathematical language allowing dynamic data types, is easier to interpret than to compile to its machine language representation. Hassit's design translates APL source to an intermediate form before interpreting it.

AEROSPACE processors: Neilson in 1973 [CHUY75] reported an aerospace processor to implement a subset of SPL (Space Progamming Language) derived from JOVIAL.

SNOBOL processor: Shapairo in 1972 [CHUY75] began a design for SNOBOL. Since SNOBOL is a string manipulation language, and the von Neumann type architecture is very unsuitable for such languages, huge preprocessing was required.

HYDRA processor: Meferland in 1970 [MCFA70] reported a HYDRA processor to support the high level language TPL. TPL is similar to EULER and has rich data structure capabilities.

Altough all the above machines are known as high level language machines most of them actually have an intermediate phase (language) for execution.

Solving architectural problems by designing high level language machines did not attract much interest, perhaps for the following reasons:

1. Though the compilation time is saved, interpretation is complex, partly because during interpretation values are not bound to identifiers.
2. High Level Language Machines become too strongly biased towards the particular language for which they are designed.
3. Not encoding the identifiers in the program makes the executable representation large.
4. Large programs will not run because of the inefficient techniques used in the

hardware implementation.

Interpretive languages such as APL, BASIC, LISP have been found to be better suited to for such machines than sequential machines. Functional languages are being developed as another viable alternative. They have not been proved to be good so far but Backus *et al.* are still working on them.

1.7 Intermediate Languages

Between the high level language and the host level language lies a range of possible program representations.

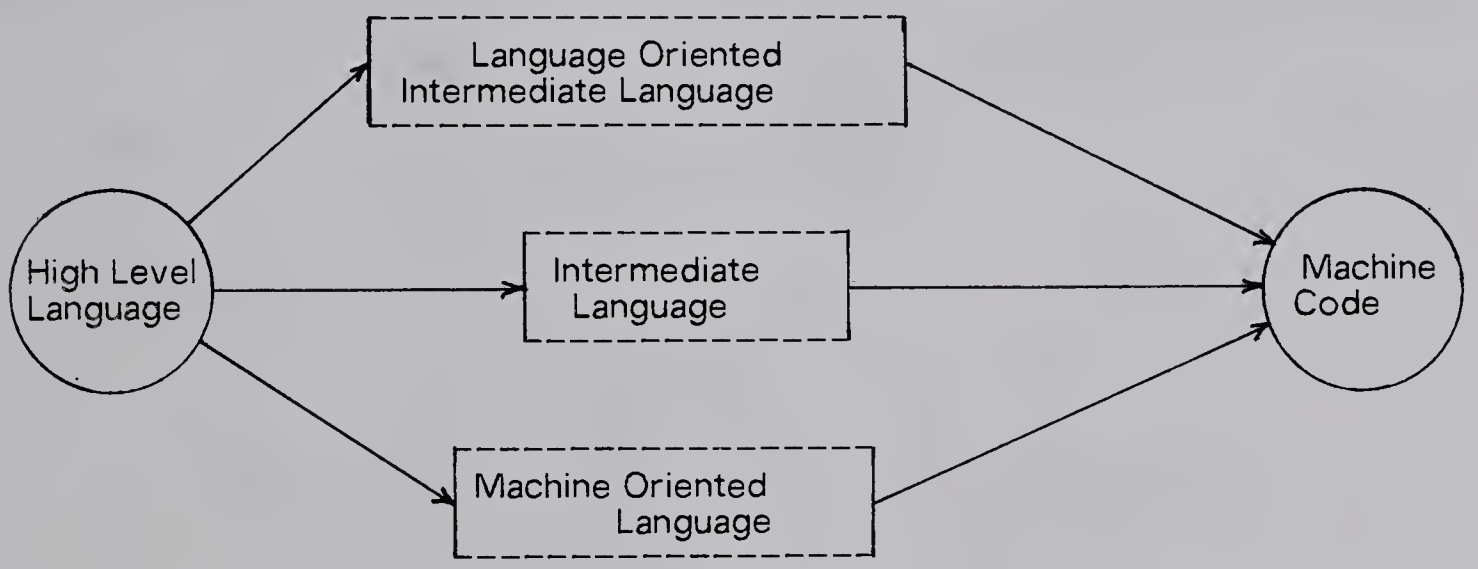

Figure 1.3. Range of program representations.

An ordinary compiler makes several passes, each pass giving an intermediate form, before producing the binary machine language equivalent of the source program. Compilers can be designed to generate intermediate forms as desired.

Such intermediate forms have been obtained earlier for various reasons. An important reason was to split the job of compiler writing into two phases. Another reason was portability. Intermediate languages were also helpful in implementing a new language on a machine or rewriting only one section a compiler. Originally, the last reason

was considered most important.

The first formal study of intermediate languages can be traced back to 1958 when a committee was set up to design a universal intermediate language UNCOL for any problem oriented language (POL)[STRG58]. This was an attempt to partition the job of compiler writing and save half the time when implementing any new language on a machine. The idea was to translate every language into a machine independent intermediate language which could then be compiled into machine code for any machine.

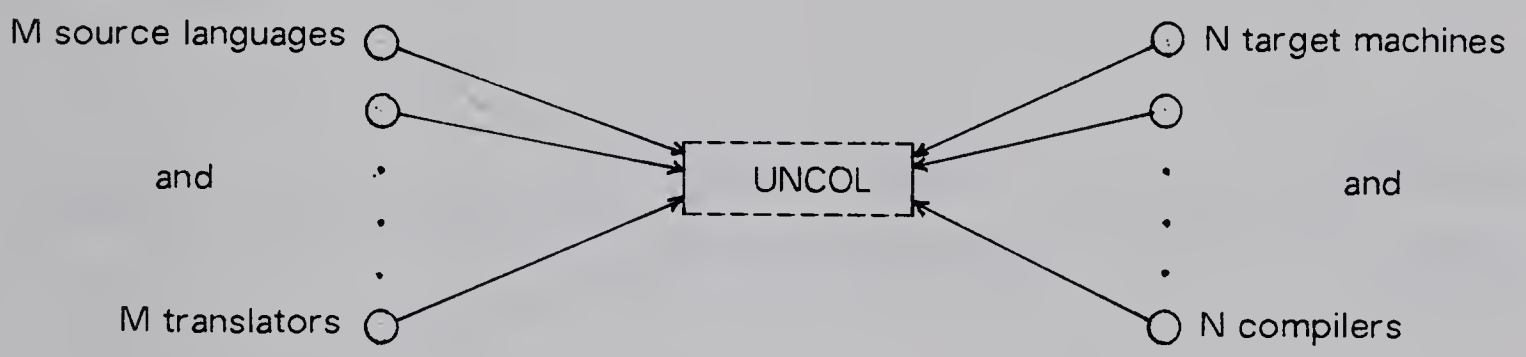

Figure 1.4. Language translation; any language to any machine.

If there were I source languages and J target machines this method would reduce the number of translators to be written from (M * N) to (M + N).

The project did not meet with much success because there is too little in common among languages and machines. It was realized that some restrictions on either machine or language were necessary. The consensus was to give up on the universality and select langauges and machines with common features.

Apart from issues of reducing the amount of compiler writing and of portability, intermediate languages were also built for interpretation or direct microprogrammed execution. Some intermediate languages used for direct microprogrammed execution are:

1. The intermediate language CTL used for ALGOL 60, FORTRAN, and PL/1 for execution on MU5 [CAPO72].

2. The OCODE generated as an intermediate language from BCPL [RICH71].
3. The ZCODE from ALGOL68C [ELSW79].
4. The intermediate language of ALGOL60 [RAND64].

There are two other forms of intermediate languages. The language oriented and machine oriented forms are discussed in some detail below.

1.7.1 Machine Oriented Intermediate Languages

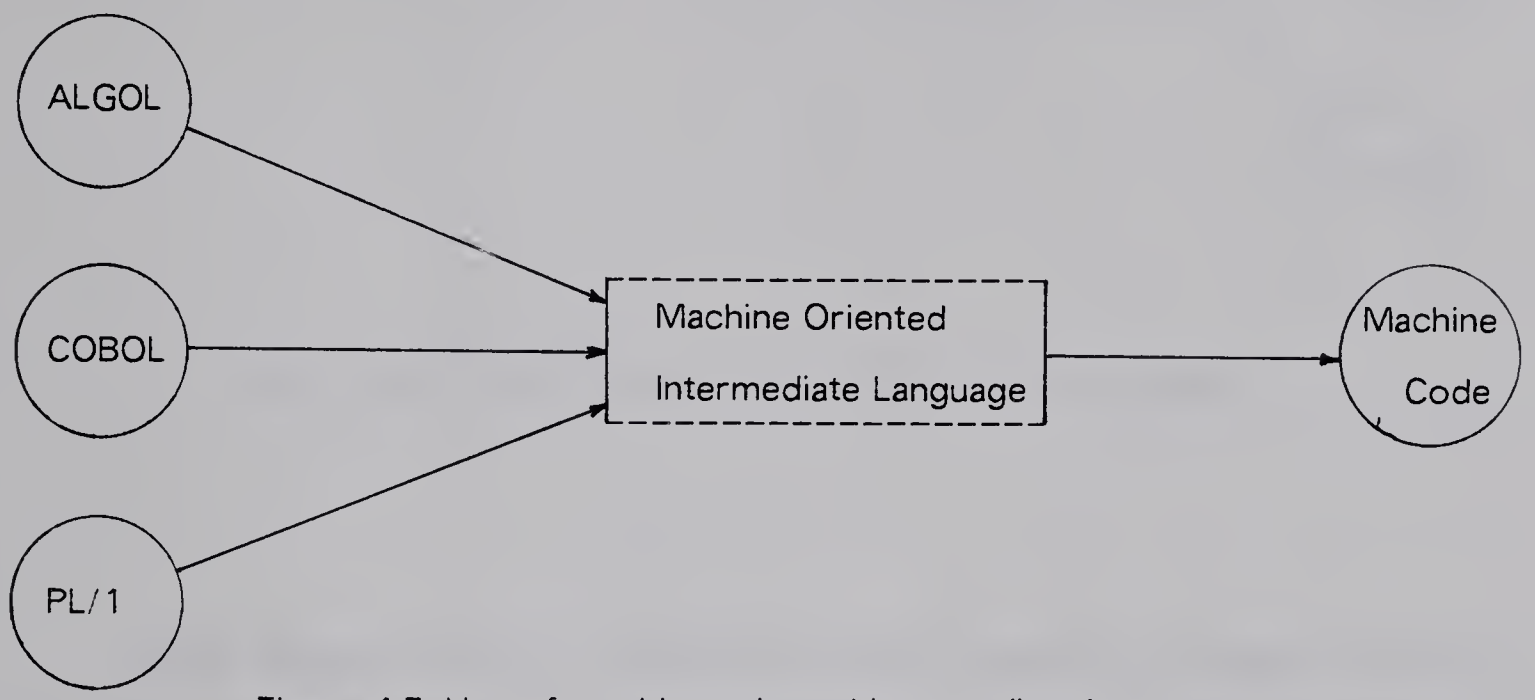

Figure 1.5. Use of machine oriented intermediate languages

A machine oriented intermediate language is often used to partition the job of compiler writing especially to avoid rewriting entire compilers to implement several languages on one machine. However, compiling a source program into a machine oriented intermediate language is more difficult than translating it into a language oriented intermediate language. A machine oriented intermediate form is easy to execute on a class of similar architectures.

1.7.2 Language Oriented Intermediate Languages

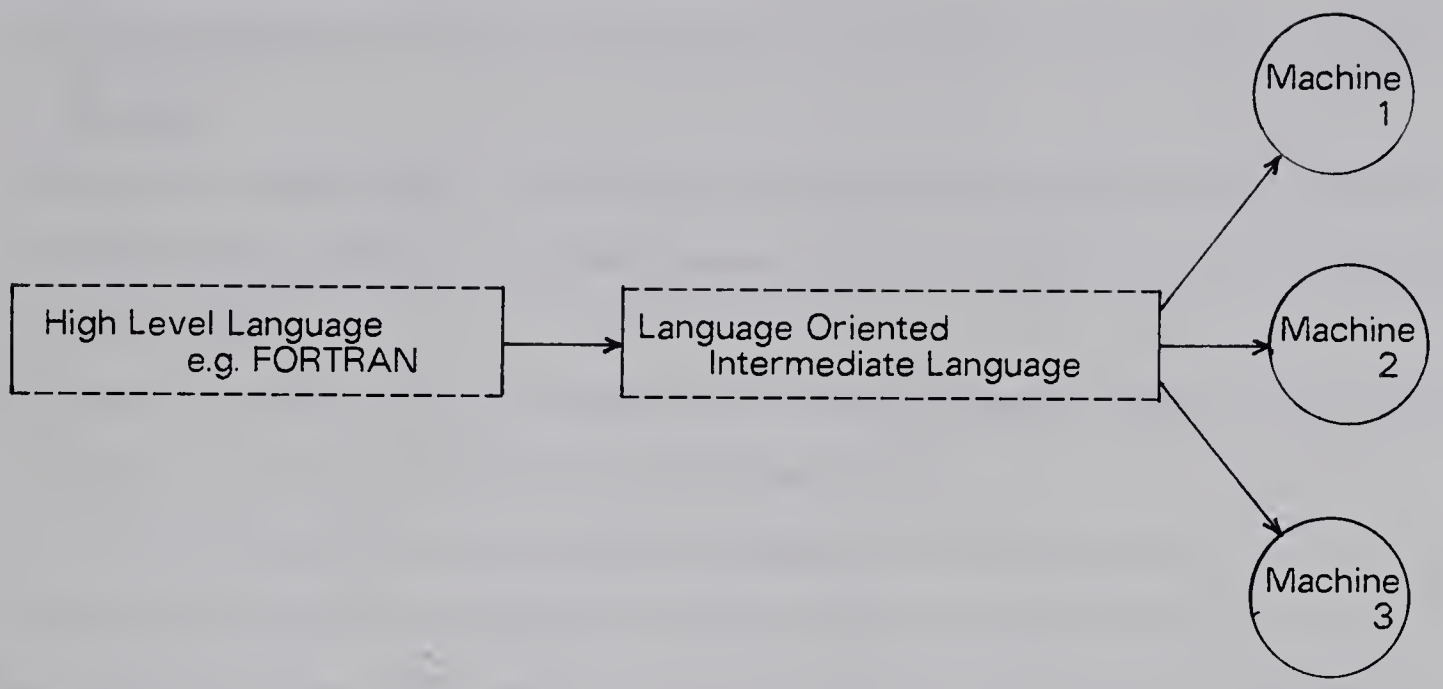

Figure 1.6. Use of language oriented intermediate languages.

Hoevel analysed the time requirments of three models of program execution environments [HOEV74]. He found that interpreting a language oriented intermediate languages was the most efficient method of executing programs. Such intermediate languages have been classified by him as DELs, for Directly Executed Languages. A DEL is defined as a program representation that is close to the high level language representation. It is a compact representation in which, for each operation in the source, there is at most one corresponding operation in its DEL representation. The symbols are in a one–to–one correspondence with the object names used in the source program. The language oriented Intermediate language is obtained by only one or two compiler passes. It therefore is simple to compile but complex to interpret.

Earlier work on language oriented intermediate languages was aimed at enhancing portability. Some of this work was:

1. BCPL was compiled to a machine independent intermediate language[RICH71].
2. PASCAL to machine independent P–CODE [ELSW79].
3. ALGOL 68C to machine independent Z–CODE [ELSW79].

Language oriented intermediate languages, particularly those developed for direct execution, have some useful properties, such as:

1. The intermediate form is close to the high level language representation. This aids in debugging.
2. For every action (ADD, SUBTRACT) in the source program there is only one corresponding action in the intermediate form. This results in fewer overhead instructions.
3. There is a one-to-one correspondence between object names in the source program and object names in the intermediate form.
4. The intermediate form is a concise representation of the program.
5. The order of operation in both the source and the intermediate forms are same.

Broadly speaking, these forms, when executed on a 'soft architecture'[2] corresponding to the language, are efficient because:

1. The inefficiencies of compilation are avoided by keeping the intermediate level closer to the language.
2. The inefficiencies at the machine level are overcome by interpreting the intermediate form on a 'soft architecture' suitable for the language.

1.8 Universal Host Machines

With the advent of microprogramming it has been possible to implement 'soft' architectures, defining a machine suitable for the language on the host. The ease and efficiency of such an implementation depends on the architecture of the host machine. Some machines are designed to allow the emulation of several machines. Such machines are refered to as *universal host machines*. Some of the machines which might be classed as universal host machines are the Varian 73, Burroughs B1700/B1800, Nanodata QM-1, and MLP-900.

The Varian 73 has a very complex horizontal microinstruction format which is 64 bits wide.

---

[2] A soft architecture is a user microprogramable architecture eg. B1700 such that the instruction set for the programmer has to be defined by microcode.

The Burroughs B1700/B1800 is a unique kind of machine with no instruction set of its own. Though it has not been very successful commercially it is of great interest to machine designers. It was designed so that the soft architecture for any particular language can be put on it before executing it thus making execution very efficient. In contrast to Varian, B1700 has a 16 bit wide vertical microinstruction. Varying size opcodes and bit addressability are among the unique features. Programs are not compiled but rather translated to an intermediate S-language and interpreted through microprograms. This machine can be classified as an intermediate language machine.

However, there are other problems with interpreting the S-language on Burrough's B1700/B1800. The decoding of instructions, though claimed to be efficient, is not clearly defined in the available literature, nor is the bit addressing overhead given. The user must load the corresponding interpreter before executing a program in a given language.

The microprogramming techniques of the Varian 73 and Burroughs B1700 have been combined in the Nanodata QM-1. The QM-1 uses two level microprogramming (micro and nano). An 18 bit vertical format is used to access a 360 bit horizontal microinstruction. This double format and the unusual word length of 18 bits make the QM-1 very flexible. Many machines including the PDP-11 have successfully been emulated on it.

1.9 Tagged Architectures

A major step in closing the semantic gap and deviating from the classical von Neumann model has been to make all data words within memory identify themselves. In a von Neumann architecture, the types and lengths of data are defined by the instructions. An attempt to deviate architectures from von Neumann type architectures was made by introducing tags.

Tags were originally suggested by Illiffe [ILLI68], for data and address descriptions, in the design of his Basic Language Machine (BLM). Illiffe's approach was to reduce the number of instructions by specifying data types with the data instead of with the instruction. Tags can detect errors such as adding a real and a floating point number. By setting some tags as 'escapes', protection and invalid addressing can also be handled.

The advantages of tagged architectures and their usefulness in the production of software for compilers and operating systems, in debugging, in register allocation and in parallel processing are discussed by Feustal [FEUS73].

Not much importance has been given to tags by manufacturers, but similar fields have been used for purposes such as specifying length of data in the Burroughs B1700 and specifying the parity in the IBM 7040.

One recent design using tags is the SWARD machine designed by Myers [MYER78]. In SWARD's architecture tags have different lengths and formats. The first four bits are always fixed to define the type of storage area.

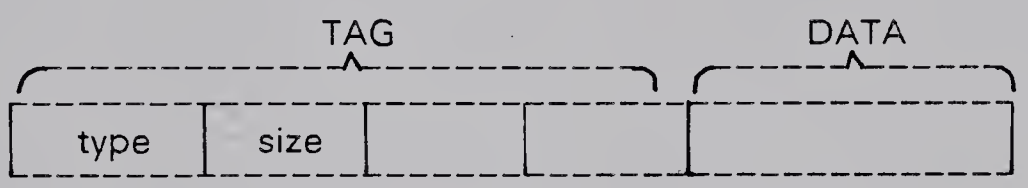

Figure 1.7. Tag format in the SWARD machine.

The only apparent disadvantage to tags seems to be the extra storage required to specify the tags. Myers gives specific examples to show that the extra storage requirement is not necessarily true.

Tags are the first step in designing machines where data and instructions are distinguishable, and should be useful in closing the 'semantic gap'.

1.10 Conclusions

Solving problems efficiently has always been difficult. With the many different kinds of programming languages it has been difficult to design a single machine which can reasonably match all kinds of applications. An expensive study to establish the quantitative and qualitative evaluation of architectures was made by the CFA in 1977[FULL77].

Until machines are designed to suit programming languages, directly interpretable representations are the only alternative. Such representations have shown substantial improvements in time and space requirements. Functional languages are also being designed. Recently in [FLYN80] it has been shown that directly interpretable forms, as compared to machine language level representations, are also efficient for distributed processes. The ultimate goal is to provide an efficient representation for every individual program, and some work in this direction has been done by Saunders[SAUN79].

2. Analytical Conditions For Interpreting Programs

Running programs through an intermediate language or machine language representation is a two phase process. Phase one is to transform the source program into its intermediate language or machine language representation and phase two is to execute it. The conventional method of running a program is to compile it to the machine language equivalent and then execute it. A viable alternative to this is to translate the source program to an intermediate form and then interpret it. The basic difference between these two approaches is that the machine language representation is dynamically short (does not require many machine level instructions to execute one representation instruction) but is statically long (requiring a lot more storage). On the other hand language dependent intermediate languages are dynamically long and statically short. Dynamically long representations require time to decode, but since fewer instructions are fetched the storage access time is small.

This chapter analyzes and compares the time required for executing programs by these two methods, compilation plus direct execution versus translation plus interpretation.

A *Directly Interpretable Representation* is defined as an intermediate language representation of the source program which can be interpreted on the host without being compiled. It is designed to make the execution phase efficient by reducing the time to fetch instructions and reducing the number of overhead instructions.

2.1 Directly Interpretable Representations

The size and form of a directly interpretable representation (DIR) makes it distinct from both a source program and a representation in machine language. The size of the directly interpretable representation can be made smaller than that of the source program because:

1. Variables in high level programming languages use symbolic names, which take a lot of space to store. Encoding them in binary reduces this space.
2. Arithmetic expressions can be replaced by parenthesis–free forms (like reverse polish) to save space.
3. Symbol table look–ups to find values can be avoided by using 'hash' techniques.

Directly interpretable representations are also distinct from the machine language equivalents of source programs. When a source program is compiled into machine language instructions, several machine language instructions, requiring more space to store, are needed to implement one source language operation. Since accessing these machine language instructions is time–consuming, execution is slowed down. If instead of this static expansion the source program can be encoded and expanded dynamically, execution speed would be increased.

The following section summarizes earlier analytical work done to show that executing programs through a language dependent intermediate language is more efficient than executing machine language representations.

2.2 An Analytical Argument for Interpreting

Hoevel [HOEV74] evaluated and compared the time required for executing programs in a two phase processing system(Figure 2.1). The two processes required are translation, or compilation, and interpretation, or direct execution. The two processes are done by programs which run on a base machine. The input to phase one is the source program and the output a machine language or an intermediate language representation is the input for phase two.

In addition to assuming that the code produced (by the compiler) is 'exact' the analysis assume the following:

A machine has two levels of memory, a main store (wms) and a writable control store (wcs). The control store is a fast memory which, in addition to storing microcode can store program and data. It is analogous to cache memory. The time required to access the control store is R times less than the time required to access the main store. The access width of the control store is at least as large as that of a machine language instruction (all machine language instructions are assumed to contain the same number of bits). The access width of the main store is not greater than that of the control store.

Given different accessing speeds for the two memories, the time required for executing programs is dominated by the time to fetch instructions. Hoevel computed the time requirements for three different methods to determine the most efficient. These three methods are analyzed and compared below.

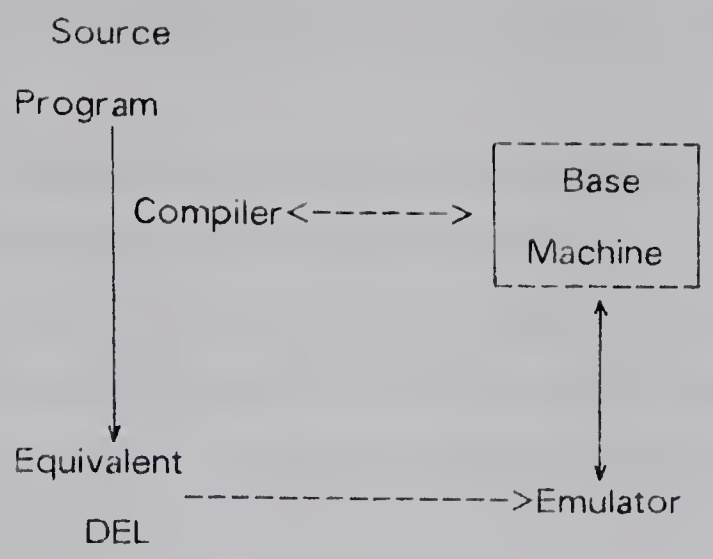

Figure 2.1 Hoevel's two phase processing system.

Method A

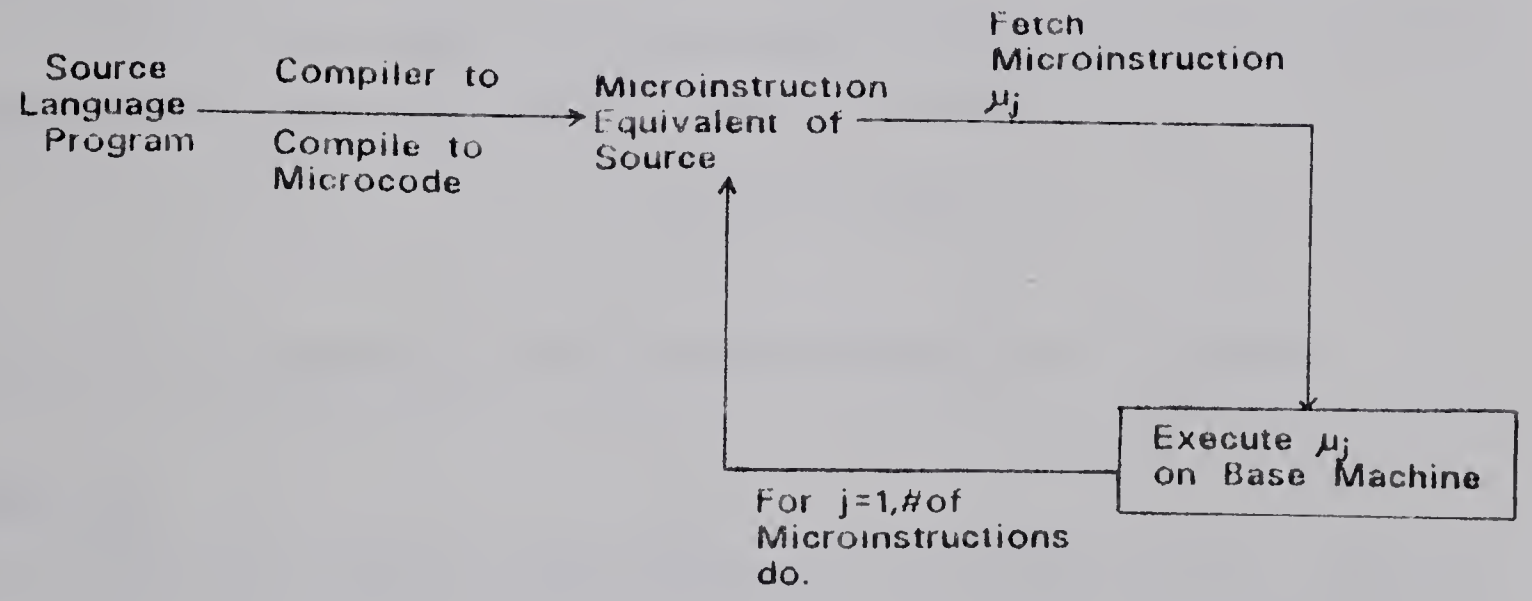

Figure 2.2. Flowchart for Method A

In this method the machine language equivalent of the source program is stored entirely in the main store and executed by fetching one instruction at a time. Time required for such execution is the time required to fetch an instruction and the unoverlapped time

required to execute it. All machine language instructions are assumed to be of the same length.

Though it is not specifically stated, the machine language instructions are the same as microcode instructions that control the gates directly and do not require any decoding or microroutines. This is a very important assumption. Compiling a source language program to exact microcode efficiently is NP-hard. Hence method A can be considered practical only if it is possible to compile the source program into exact microcode, and only if the time to fetch an instruction is not more than the time to execute the instruction, so that instruction fetching and executing can be done in parallel.

The time required for method A is computed as follows:
If wmi is the width of machine language instructions, wms the width of main store and R the cycle time to fetch from main store, then the time required to fetch a machine language instruction from main store is wmi/wms * R. (It is assumed that wmi is an multiple of wms.)

If N is the total number of machine language instructions executed, then the time required for the execution of program is computed as follows.

$$T(A) = N (wmi/wms * R + EM)$$

where EM is the unoverlapped time to execute all instructions from main store.

Method B

In this method the machine language equivalent of the source program is executed through the control store which, is assumed to contain blocks of executable code in addition to the interpreter.

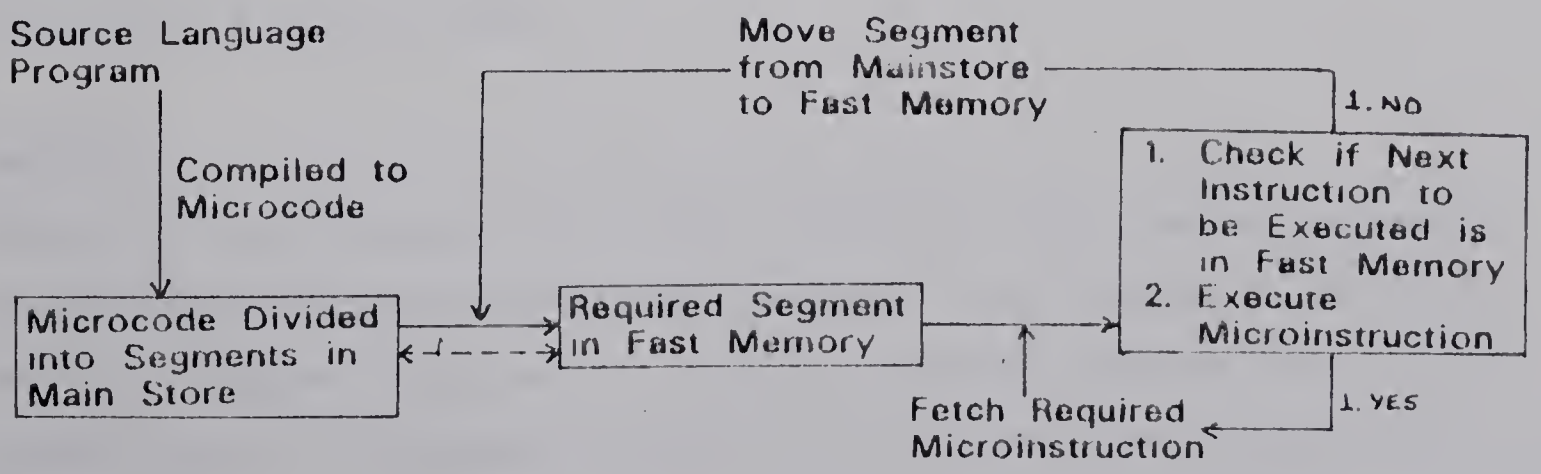

Figure 2.3 Flowchart for method B.

Since the machine language equivalent of the source program is long and cannot fit into control store, it is stored in main store and blocks of code moved into control store as required. Individual machine language instructions are fetched from control store, which takes much less time than fetching from main store, and executed. Execution continues until the required instruction is not in control store. At this stage a fault is said to have occurred, execution stops and the required block of code is moved into control store.

The time required for the execution of all instructions from control store is

$$N (L + wmi/wcs + EC)$$

where N is the total number of instructions executed, L is the time to locate a machine language instruction in control store, wcs the width of control store, wmi/wcs the time to fetch an instruction from control store, and EC the unoverlapped time to execute all instructions from control store.

The time to move a block from main store to control store when a fault occurs is

$$Tbloc + wbloc/wms * R$$

where Tbloc is the time required to locate the block and wbloc width of the block.

Hence, the total time required to execute the program is

$$T(B) = NF* (Tbloc + wbloc/wms * R) + N* (L + wmi/wcs +EC))$$

where NF is the number of faults.

Method C

The previous two methods require a non-trivial compiler. Another method of executing source programs employs a trivial translator and a non-trivial interpreter. In this method the source program is translated into a directly interpretable representations. A trivial translator keeps the intermediate program form close to the source language. Such an

intermediate program form will be termed a high level directly interpretable representation (DEL–directly executable language in Hoevel's terminology) for the rest of this thesis.

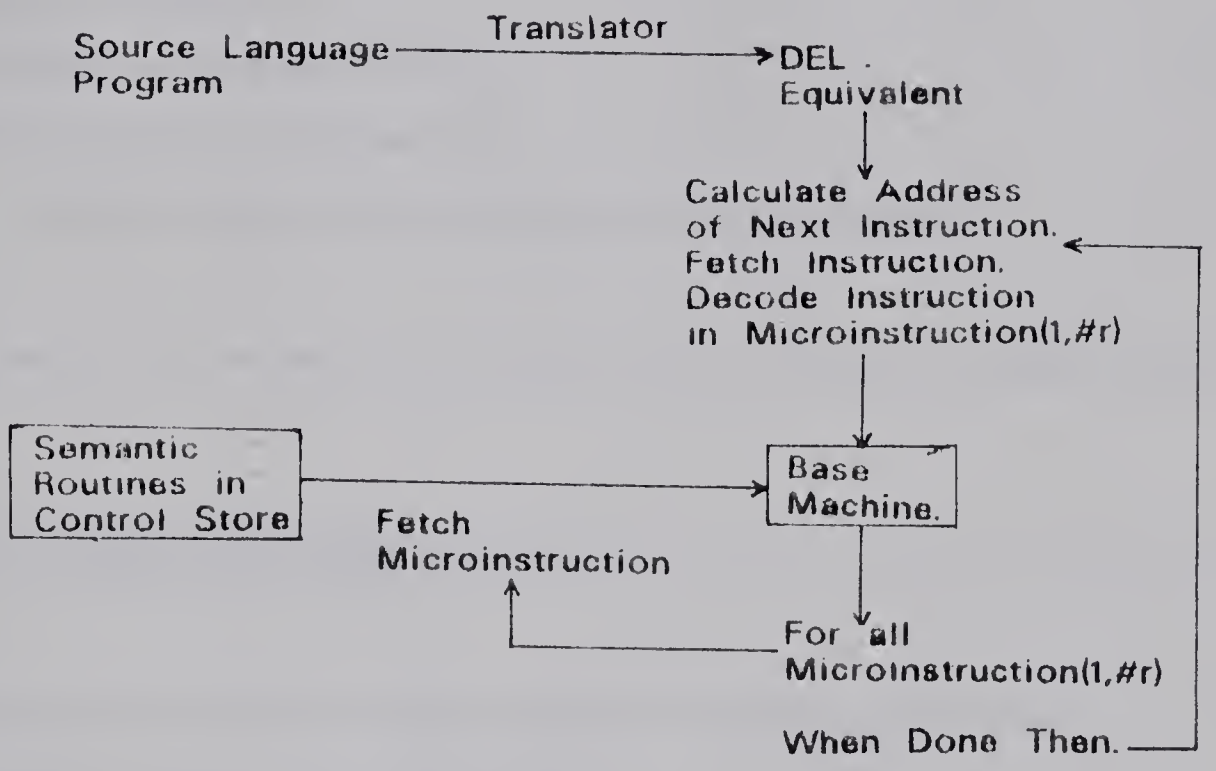

Figure 2.4 Flowchart for method C.

The interpreter determines the address of the instruction to be processed, fetches the instruction, decodes it and initiates the corresponding microroutines. If TADDR is the time required to determine the address of the next high level DIR program instruction to be executed, DECO the time to decode, wdi the width of high level DIR program instruction, then the time required for the execution of an entire DIR program is

$$T(C) = ND * (TADDR + wd/wms*R + DECO + NM (wms/wcs + ED))$$

where ND is the total number of DIR instructions executed, NM is the average number of machine language instructions executed for each DIR program instruction, and ED is the unoverlapped time to execute all the instructions

Comparisons

The number of instructions executed N, ND, and the faults NF depend on the source program and its translators (assumed to give the best code). To eliminate these dependent variables the following are defined:

FA: 100 * NF/N the fault rate.

FI: TADDR + DECO the interpreter overhead

PH: NM − N/ND the high level DIR overhead

NE: NM − FI * (N/ND) effective machine language instructions executed.

For Method C to be faster than Method A, T(A)−T(C) must be greater than zero, giving

$$R > \frac{PH+N1*((wmi/wcs)+E2-E)+FI*(E2+(wmi/wcs))}{N*(wmi/wcs)-(wd/wcs)} \quad \text{..... (1)}$$

The denominator of the right hand side of equation 1 is always greater than 0.

For method C to be faster than method B

$$-R < \frac{PH+N*(E2-E-L-(S*FA)/100+FI*(E2+(wmi/wcs))}{NE(wseg/wms)FA/100-(wd/wms)} < R \quad \text{..... (2)}$$

depending on whether the denominator is negative or positive.

Equivalently, expressing the difference, of time rate we get required for method B and method C, in terms of the fault

$$FA > \frac{100*PH+(wseg/wms)*R+N*(E2-E1-L)+FI(E2+(wmi/wcs)}{NE(S+(wseg/wms)*R)} \quad \text{..... (3)}$$

Equation 1 gives the minimal value of R for which interpreting an intermediate language would be faster than executing machine language instructions on the processor. Under usual conditions, the time to fetch all the machine language instructions executed in a program is greater than the time required to fetch one high level DIR instruction to control store. Hence the denominator in (1) will be positive. The value of the right side of equation 1, according to the Hoevel, is less than 2 for an IBM 360, the Tucker and Flynn

[FLYN71] and the QM-1 machines. The time to access main store is more than double the time required to access control store.

Equation 2 gives the minimal and maximal values of R in buffered execution. The denominator in equation 2 is the difference between the time to move faulted blocks into control store and the time to fetch one high level DIR instruction. Unless the fault rate is 0, the denominator is positive. The term L makes the numerator negative thus making interpreting a high level DIR an efficient means of execution for any fault rate.

If R is between 4 and 16, it is argued that the numerator of the right side of equation 3 is always negative and hence method C is faster than executing machine language representations (method B) for any fault rate.

2.2.1 Criticism and Discussion

Hoevel's work is a theoretical argument to justify executing programs through high level DIRs. It does not put any condition on the size of high level DIRs but rather sets conditions on the ratio of the cycle times of main and control stores. In Methods A and B the languages considered are assembler type languages. The analysis ignores translation and compilation times, which are reported to be about ten percent of the system time. Also the code produced by the translators and compilers is assumed to be exact. It is possible for the value of PH * (NM – N/ND) to be negative in which case the argument fails.

The terms PH and FI can be determined only when all the DIRs and their interpretors are available. Hence they are not as easy to determine as claimed. It is also claimed that under current technological developments the factor L (which is the time required to locate a microinstruction in control store) would always keep the numerator of the right hand side of equation 2 negative. With the increase in the size of fast memory and the decrease in the speed of accessing, the factor L may not always make the numerator negative. Much depends on the high level DIR overhead PH. This factor is difficult to evaluate.

Though Hoevel's work is the first of its kind no definite conditions for interpreting DIRs have been determined. Considering all the variables ignored or not well defined it should be possible to determine conditions under which executing DIR

programs would be efficient.

2.3 Executing Programs from Writable Control Stores

It has been observed by Knuth [KNUT67] that more than half of execution time is spent on less than 4% of a program. Liu and Mowle [LIUM78] devised techniques to detect this small portion of the program and put it into control store, thus speeding up execution. In order to compensate for the time required to fetch a program into control store and load data into registers, a loop has to be executed a certain minimum number of times. Assuming that all operands and results occupy one memory word each the run time overhead of loading and restoring inner loops is

$$T1 = 3 * M * F1 * N + 3 * M * F2 * N + Tp * M (F1+F2) * N$$

Let

T1 be the overhead in preloading and restoring variables and operands,

Tp be the average execution time of a microinstruction in terms of main memory cycle time,

M be the main memory cycle time,

F2 be the proportion of machine instructions that require a result returned to main memory, and

F1 be the proportion of the machine instructions that require a main memory reference for each operand.

3 * M * F1 * N is the time required to preload operands.

3 * M * F2 * N is the time needed to restore results, and

T * P * M *(F1 + F2) * N is the time to execute the restoring and preloading microinstructions.

When a loop is implemented in microcode and stored in control store there is no need to fetch instructions, to store results back into the main memory, to fetch operands, or to decode instructions. This results in the saving of execution time. The total execution time thus saved is

$$S1 = M*R1*N + M*R1*F1*N + M*R1*F2*N + M*R1*D*N$$

where R1 is the number of times the loop must be executed to recover T1, S1 the total time to execute loop R1 times, D is average decoding time for one machine instruction in numbers of main memory cycle time.

The first term is the time saved by not fetching instructions, the second term is the time saved by not fetching operands or results in registers, third term is the time saved in storing results and the last term is the time saved in decoding.

The time needed to recover the overhead S1 should be equal to T1. Equating these two and taking the worst case F1 + F2 = 1

$$Rmin = (3 + Tp)/ 2 + P$$

Similarly the overhead of dynamic overlaying is

$$R3 = CL/(1 + (F1 + f2) + D \text{ or } CL/1 + D \text{ if } F1 + F2 = 0$$

where R3 is number of times a non inner loop microcode block must be executed for the overhead to be recovered, C the average number of microinstructions required to implement one machine code, L the ratio of micro and main store words.

These results give a clear idea of what sections can be microcoded and when. The results, though not directly developed for directly interpretable representations, can very easily be implemented for further gain in speed. However, this requires space in control store, which can be obtained by curtailing the size of the interpreter.

2.4 Conditions Under Which Interpreting Is Efficient

Phase one, in a two-phase processing, is translation in which the source is parsed through syntactically and semantically and a surrogate produced. Phase two is to execute the surrogate in one of the following forms:

1. if the surrogate is in microcode it is executed directly by state transitions. (hard-wired interpretation)
2. if the surrogate is in host's language it is interpreted and executed(trivial). (microprogram interpretation)
3. if the surrogate is at a higher level form then interpret it and then execute.

(interpreter interpretation).

On the other hand, if the program and machine can be brought closer together, one of the phases would become trivial. That is if the programmer expresses the program in host's instructions then translation is trivial and if the machine executes the translated code then execution is trivial since no interpretation is required.

Excluding the methods in which one of the above phases is trivial we have the following methods for comparison:

Translation	Interpretation
Compiling source to host machine's instructions	Executing host code
Translating HLL to an intermediate language	Interpreting surrogate and executing
Translating HLL to U code	Executing the surrogate by state transition

Among the practical methods to execute each of the surrogates are to keep the surrogate in a fast memory and execute taking one instruction at a time. If the surrogate is larger than the available fast memory then store it in main store and execute by moving in the required blocks to the fast memory.

Conventional Method Of Executing Programs

Method 1

In this method the source language program is compiled to host-language instructions (conventional method) and stored in the computer's main memory. Blocks of machine language instructions are brought to control store and executed by fetching one instruction at a time from the control store.

Source program ---> Compiler ---> Host instructions ---> Simple interpretation ---->Execution

The compilation is complex, resulting in large executable program representation but execution is straight forward. The sequence of steps for such an execution are

1. compile program
2. fetch blocks to control store
3. if instruction needed is in control store execute it through corresponding microprograms, else go to 2.

By a sequential technique or otherwise blocks are brought into the control store and execution continues until a fault occurs. When a fault occurs the required block has to be fetched.

Time required to move in a faulted block to control store is

$$TS + WS1 * TM + WS2 \quad \text{..... (1.1)}$$

where TS is the time required to locate a block in main store, WS1 is the (average) number of instructions in a block, TM is the time to fetch a machine language instruction from main store to control store, and WS2 is the time required to move a block from control store to main store. Moving the block from control store to main store is not necessary when there is a copy in main store, hence WS2 can be neglected.

Time to execute an instruction in control store is

$$TLC + TIF + Te \quad \text{..... (1.2)}$$

where TLC is the time required to locate a machine language instruction in control store, TIF the time to fetch a machine language instruction from control store for execution, Te the unoverlapped time in fetching–locating–executing a machine language instruction.

Hence, the total time to compile and execute is

$$TCH+TDEM+TEM+NF(TS+WS1*TM)+((NF+PS)*AVI(TLC+TIF+Te)) \quad \text{.....(1.3)}$$

where TCH is the time required to compile, TDEM the time to decode all the machine language instructions that are executed, TEM the unoverlapped time for moving predicted blocks, NF the number of faults in blocks of machine language instructions, PS the number of blocks predicted and, AVI the average number of instructions executed per

block.

Executing Intermediate Representations

Method 2

In this method the source program is translated to an intermediate representation and stored in the machine's main store. The surrogate is executed by fetching one instruction at a time decoding and executing the corresponding microroutine.

HLP ------> Translator ------->Interpreter ------> Host

All DIR instructions are fetched from main store, one by one, decoded and then executed. The time required to locate, fetch and decode a single DIR instruction would be

TFA + TFD + TDE ..... (2.1)

where TFA is the time to locate the DIR instruction needed for execution, TFD is the time to fetch a DIR to control store and, TDE the time to decode a DIR instruction.

Time required to execute each DIR instruction decoded is

TLC + TIF + Te1 ..... (2.2)

where Te1 is the unoverlapped execution time.

Thus the total time required is

TCD + NDE(TFA + TFD + TDE + NDI (TLC + TIF +Te1)) ..... (2.3)

where TCD is the time required to translate from source language to DIR form, NDE the number of DIR instructions executed, and NDI the average number of machine language (micro code) instructions per DIR instruction.

Note: for each instruction in DIR program instruction decoded a set of microinstructions are obtained which have to be executed for the equivalent operation in the DIR program.

Execution With Buffering

Method 3

In this method the DIR program is executed by transferring blocks of instructions to a fast memory. It is assumed that blocks are moved into control store in parallel to execution. The sequence can be broken by branch instructions, for example, requiring some other block to be fetched to continue execution. When a fault occurs the execution stops until the required block is fetched. Also with the parallel fetching of predicted blocks there could be a wait time associated with moving in the blocks.

The time required to move all the faulted blocks into the control store is

NF2 * (TDS + TFD * WDI) ..... (3.1)

where NF2 is the number of faults, TDS is the time to locate a block, and WDI the width (height) of a block.

If TDO be the unoverlapped time for moving blocks then unoverlapped time for transfering blocks is

PS*TDO ..... (2.3.2)

giving the total time to translate and execute as

TCD+PS*TDO+NF2(TDS+TFD*WDI)+(NF2+PS)*NDS(TDL+TDE+NDI(TL+TIF+Te))

where TDL is the time required to locate a DIR program instruction in control store, and NDS the number of DELs per block.

Execution By Compiling Source To Microcode

Method 4

In this method the source program is compiled directly to host microinstructions producing microcode which can be directly executed by the host.

High Level Language ------> Compiler ------> Host code

Generating efficient microinstructions that control all the internal parallelism is known to be an NP-hard problem and hence compilation would be very complex. Execution time would be minimized depending on the efficiency of code generated, but the resulting program would be very large. This method could be advantageous if the source program is executed many times and the compile time and the time to interpret are the dominating factors.

Time to compile and execute by fetching one microinstruction at a time to control store would be:

$$TCOM + (TLM + TFM + TIF + Te2)$$

where TCOM is the time required to compile to microcode, TLM the time required to locate microinstruction in main store, TFM the time to fetch a microinstruction from main store, and, Te2 the unoverlapped time to fetch-execute instructions.

Executing Microcode with Buffering

Method 5

In this method the microcode of Method 4 is executed by transfering blocks into control store. As in method 3, blocks are moved into control store and executed. The time for moving in 'faulted' blocks is

$$NF3\ (TS2 + WS3*TM)$$

where TS2 is the time to locate a block of microinstructions in main store, NF3 the number of faults for microcode blocks, and WS3 the width of a micro code block. Time to locate, fetch and execute a microinstruction from control store is

$$NC * (TLC + TIF + TFE1)$$

where TFE1 is the unoverlapped time to fetch and execute.

The total time required is

$$TCOM+NF3(TS+WS3*TM)+PS(Te3)+(NF3+PS)*NIS(TLC+TIF+TFE1)$$

where Te3 is the unoverlapped time for predicted blocks, NIS the number of instructions per block, TFE1 is the unoverlapped execution time.

2.4.1 Comparison And Discussion

Comparison of Methods 1 And 2

For execution by fetching one DEL program instruction at a time (method 2) to be faster than execution with blocks of host language instructions (method 1) we must have

$$(TCH+TEM+TDEM+NF(TS+WS*TM)+(NF+PS) * AVI(TLC+TIF+Te)) - (TCD+NDE\ (TDE+TFD+TDE)+NDE*NDI(TLC+TIF+Te1)) > 0$$

The part for difference in time to execute the executable representation from the control store is:

$$((NF+PS)*AVI(TLC+TIF+Te))-(NDE*NDI(TLC+TIF+Te1)) > 0 \quad \ldots\ldots (5.1)$$

(NF+PS)*AVI is the total micro instructions executed from the compiled version, and NDE*NDI the total micro instructions executed through the DIR. Since the DIR program representation is compact and small the condition is always true, and the difference would depend on the size of DIR program, and the efficiency of interpreter.

The remaining part is the time for fetching the executable representation (micro code) to the control store.

$$(TCH+TDEM+TEM+NF(TS+WS1*TM))-(TCD+NDE(TDE+TFD+TDE)) > 0 \quad \ldots\ldots (5.2)$$

From the above, the difference in time to compile is

$$TCH + TDEM - TCD > 0, \text{ or } TCH + TDEM > TCD.$$

The inequality is always true because the time to compile to host instructions is always greater than the time to translate to a DIR (DIRs can be obtained with one or two passes of the compiler, whereas compilation needs more) program.

The remaining part of the equation is

$$NF * (TS * WS1 + TM) - NDE\ (TDA + TFD + TDE) > 0$$

or,

$$NF * (TS * WS1 + TM) > NDE\ (TDA + TFD + TDE)$$

For this to be true the size of the DIR program should be small which makes NDE small and there should be sufficient parallelism. In the worst case the number of faults could be zero in which case the other two differences should be large enough to make interpretation efficient otherwise method 1 would be superior to method 2.

$$NF > \frac{NDE * (TDA + TFD + TDE)}{(TS * WS1 + TM)}$$

This factor determines the number of faults for which interpreting would be faster than executing through compilation. In case NF is close to 0 the execution of DIR will have to be parallel enough to make the denominator negative. Another possibility is to make 5.1 large enough to compensate for this difference. It is apparent that a major factor in making the DIR faster than conventional execution is to make the difference 5.1 larger and this can be done by having a compact DIR program, and interpret it efficiently. It can be concluded that direct execution of a DIR program would be efficient when the fault rate in executing the buffered host instructions is high and the difference in 5.1 and 5.2 is large enough to compensate for 3. The right hand side of 5.3 depends on the parallelism in the machine. If the machine is sufficiently parallel then even very few faults can make interpretation faster.

Execution of blocks of DIR Versus blocks of Compiled Version

For method 3 to be faster than that of method 1 the difference in time of method 3 and method 1 should be greater than 0.

$$(TCH+TDEM+TEM+NF\ (TS+WS1*TM) + (NF+PS) * AVI\ (TLC+TIF+TE)\ ) \cdot - (TCD+PS * TDO+NF2\ (TDS+TMD*WDI) + (NF2+PS) * NDS(TDL+TDE+NAI\ (TL+TIF+TE))) > 0$$

Analyzing the above equation we get the difference in time to move all the required blocks to the control store as

$$(TEM+NF(TS+WS1*TM))-((PS*TD0)+NF2(TDS+TMD*WDI) > 0$$

Since block size can be adjusted to have overlapped transfer and execution, TEM and (PS*TD0) can be neglected giving

$$NF(TS+WS1*TM) > NF2(TDS+TMD*WDI)$$

or

$$\frac{NF}{NF2} > \frac{TDS+TMD*WDI}{TS+WS1*TM} \quad \text{..... (6.1)}$$

Hence, the ratio of the number of faults should be greater than the ratios of the time to move in a block. This is always true !

The difference in time to decode the source program to microcode is

$$((TCH+TDEM)-(TCD+(PS+NF)*NDS(TDL+TDE) > 0$$

$$TCH+TDEM > TCD+(PS+NF)*NDS*(TDL+TDE) \quad \text{..... (6.2)}$$

$$TCH+TDEM-TCD > (PS+NF)*NDS(TDL+TDE)$$

The difference in time to compile and decode to microinstructions and to translate the DIR program should be greater than time to locate and decode all the executed DIR program instructions. This is difficult to achieve unless the number of DIRs actually executed is very small, and there is good parallelism. But in method 1 compilation time is the time spent on compiling the complete program whereas in decoding DIRs only the required instructions are decoded which saves time.

The remaining portion of the expression is the time to execute the microcode in the fast memory.

$$((NF+PS)*AVI(TLC+TIF+Te))-((NF2+PS)*NDS*NAI(TLC+TIF+Te))>0 \quad \text{..... (6.3)}$$

Since number of microinstructions executed through DIRs is less than the machine language instructions executed equation 6.3 is always true.

Conclusion: 6.1 and 6.3 can easily be more than 6.2 but for a large number of runs all other factors excepting TCH (time to compile) get multiplied by the number of runs and the difference in 6.2 would increase. For a large number of runs the difference in 6.2 could become a dominating factor and thus make interpreting slower.

Comparison between method 2 and 4

In this section the method of compiling to microinstructions (method 4) and executing is compared to that of executing the DIR program by taking one instruction at a time (method 2). For the latter to be faster

$$(TCOM+NC(TLC+TFM+TIF+TE2)) - TCD+NDE\ (TFA+TFD+TDE+NDI\ (TLC+TIF+TE1)) > 0$$

Time required to get the microcode to the control store

$$NC(TLC+TIF+TE2) - NDE*NDI(TLC+TIF+TE1) > 0 \quad \text{.....} \quad (7.1)$$

and

$$NC-NDE*NDI > 0$$

Hence 7.1 is always true. The remaining part is the difference in time to have the executable representation to the control store is

$$TCOM+NC*TFM-TCD+NDE(TFA+TFD+TDE) > 0$$

$$TCOM-TCD > NDE(TFA+TFD+TDE)-NC*TFM \quad \text{.....} \quad (7.2)$$

The right hand side of 7.2 can also be negative for one of the following reasons: 1) NC, the number of micro instructions, is larger than NDE the number of DIR instruction 2) the corresponding fetching time is high. With the left hand side positive and right hand side negative 7.2 is always true. This equation will be false when the progam is executed many times in which case it has to be compiled only once.

2.5 Conclusions

As long as 7.2 is true, interpreting would be faster. But, as in the previous case, when the number of runs is very large then the decoding time could become very large and increase the total interpreting time.

2.6 Discussion of Results

Compact representation of the executable code is space saving. It is also time saving because it saves time to fetch. Therefore the representation should be compact and compaction should be done such that the decoding is not very complex.

3. High Level Directly Interpretable Representations

The term Directly Executed Languages(DEL) was coined by Flynn and Hoevel in 1974. Though directly executable forms of intermediate languages were in existence earlier they were not particularly designed. They originated as intermediate forms in phases of compilation.

After studying the architecture of the IBM 7090 and the IBM 360 Flynn suggested design approaches for high level DIRs (DELs in Flynn's terminology). His goal was to provide an efficient translator producing an efficient output (HLDIR) which becomes an efficient input for the interpreter. His design characteristics were aimed at making every step in program execution efficient, because inefficiencies introduced while translating cannot be overcome when executing.

Rau [RAUR78] evaluated various forms of program representations. He classified the representations as follows:

High-Level Representation: the highest on the spectrum of representations from the source language program to its equivalent microcode. The High level representation is the high level language version written by the programmer.

Directly-Executable Representation: The directly executable representation is the lowest level of representation on the spectrum mentioned above. It is the mapping on the host. The DERs are further classified into S-DERs, P-DERs and C-DERs. The representation that is the lowest semantic level is classed as S-DER. This level gives the shortest execution time. If a customizing technique, similiar to that of macros, is used to call lines of code with proper parameters the resulting code will be compact but not as fast (as S-DER). This customized representation of S-DER is termed as P-DER.

An extension to this compaction method is to combine a sequence of procedure calls and replace them by one single call. The resulting representation is termed as C-DER.

Directly interpretable representation: This is an intermediate form between the source language level and the machine code level. It can be viewed as an encoded form of Directly Executable Representation. An example of the directly interpretable representation is the directly executable version of FORTRAN called DELTRAN.

Assuming execution of the programs to be done in an environment in which the machine language is in the middle of a spectrum with microprograms, nanoprograms, emulators on one side, and translators, compilers, interpreters on the other it can be seen that

S-DER is the least compact but fastest to execute.

P-DER/C-DER reduces the memory access time but requires efficient representation on either side of the line (middle of the spectrum) such as an encoding easier to decode by the interpreter on the other side.

3.1 Need For DIR's

The space and time complexity of executing a high level language directly or its machine language equivalent suggests that an intermediate version should minimize the inefficiency.

The intermediate language representation between a high level language and host instruction set should be such that it eliminates

1. the large number of temporary local storages required for organisational purposes.
2. the complex representation of arithmetic expressions. Lawson [LAWS68] suggests that representing arithmetic expressions in reverse polish is efficient for execution.
3. the complex sequencing within programs
4. the mismatch between data structures in the source program and in the host which usually results in complex addressing.
5. the overhead instructions required to execute a user's instruction (for example moving data to local storage such as registers before performing an operation).
6. the large number of memory references that become necessary when a large set of data is required by a few instructions and not enough registers are available.

The above are some of the factors that can make execution through DIRs fast. A solution to all the problems listed above is to reduce the semantic gap between the language and the host. One way of reducing this gap is to bring a particular language and machine close together which on the other hand takes it (i.e., machines) away from the others(i.e., languages). Another solution is to reduce the language to an intermediate level and implement a soft architecture i.e., an instruction set suitable for the intermediate language

on the machine through microprogramming.

3.2 Characteristics Of DIRs

Efficiency of a DIR depends on a number of factors some of which are the following:

1. It should be both time and space efficient for its translator. The code generated should be efficient and independent of host's architecture.
2. It should be an efficient input for the interpreter. Arithmetic expression evaluation techniques should be efficient, and have easy-to-decode formats.
3. It should have a close correspondence with the source program to facilitate debugging.

The output from the translator could be language-dependent or machine-dependent. Machine-dependent code requires a relatively complex compilation and a nontrivial interpretation. Machine dependent intermediate form is close to the host instructions representation as in the traditional method of execution. Also when the translation process is complex, inefficiencies are liable to go unnoticed. The problem of allocating the resources such as registers is NP-complete. Overhead instructions such as LOAD and STORE make the representation more inefficient. This level of program representation is not advantageous for portability either [ELSW76].

At the other extreme the output from the translator can be a language-dependent intermediate form. This is a concise program representation which also corresponds to the source closely. It is analogous to assuming that the high level language is the image machine and the DIR its host code, when the DIR is interpreted by a soft architecture suitable for the language. The order of operations in a language-dependent DIR should be the same as in the source program for ease of debugging.

The output from the translator, that is the DIR, is the input to the interpreter. The interpretation time can be minimized by using an efficient DIR and an efficient interpreter. As we have seen, in chapter two, most of the interpretation time is spent on fetching 'data' to be interpreted. If the program size is kept to a minimum fetch time during interpretation will be reduced. The size of the DELTRAN (without using techniques to reduce code on the basis of probability) has been shown to be about five times smaller

than the size of a conventional (host instruction) representation. This size can be further reduced with a very small increase in complexity of interpretation by considering the probablities of occurences of the operations and operands.

The well known coding technique of Huffman is complex to decode. A technique which is claimed to give a compaction that is about one percent longer than Huffman's but fourteen percent faster to decode has been suggested by Wilner [WILN72]. This scheme has been used in Burrough's B1700 and shown to save about fifty percent of the space. The 4-6-10 form implemented on B1700 uses 4, 6 and 10 bits for operation codes depending on their probablity of occurence. Some of the bit combinations in this scheme are used for 'escaping' to read more bits and get the complete instruction.

The efficiency claimed by Wilner has been challenged by Hoevel and Flynn [HOEV77] who suggested another scheme. Their scheme requires two registers and a sequence of zeros to indicate the end of the information in a word. Codes are assigned to instructions such that frequently occuring codes have a greater number of trailing zeros. If M bits are required to be packed into the remaining N bits of a word (M > N) it is possible to pack the N bits if the trailing M-N bits are zeros. The decoder can still identify the instruction. But no data about its working is available.

Wilner's scheme is ideal for bit machines such as Burrough's B1700 and is also straight forward to implement whereas Hoevel's is practical for word and byte machines too. Schemes to encode are easier to develop and implement. Encoding schemes that save upto seventy five percent have been developed [HEHN76]. But decoding is a complex problem. Also word/byte machines offset the savings by word/byte alignments. The only solution to the decoding problem seems to be in developing suitable hardware.

3.3 Instruction Formats

Instruction formats are used in all directly interpretable and assembly languages to specify the order, sources, destinations and types of operands. The format of an instruction can be specified independently or within the instruction itself. Most of the popular architectures use fixed formats which results in the necessity of having many different representations of every instruction.

An example of fixed format instructions is the IBM 360/370 instruction set.

The IBM 360/370 assembler language has six formats. They indicate the source of operand addresses(registers, instruction itself, or main memory with displacements). These format types indirectly indicate the instructions length within the instruction. IBM's formats are identified with the instruction itself. For example, an RR type instruction indicates that two numbers denoting registers are the addresses of source operands and also that the size of instruction is two bytes. An SS type instruction indicates that the source of operands is main store and that the instruction is eight bytes long. This type of formatting is also used by many other assembler languages.

In IBM 360/370 machine language a choice of formats is encoded in the instructions themselves and hence no extra space is required. However, this method requires a different instruction for each combination of source operands. For example, there are fifteen types of ADD instructions in the 360's instruction set. If another important facility such as a stack were added, more instructions would be needed. It is possible that because of this only 20 instructions are being used for more than 70% of the time.

Formats within the instruction increase the length of the instruction and the total number of instructions. This indirectly increases the size of interpreter (as more semantic routines would be required) in interpreting machines.

Earlier, intermediate languages did not deviate much from this type of formatting. The directly executable form of the high level language EULER also had six types of formats. The formats in its intermediate strings indicated operand types such as real, integer, reference and label. Since this language was developed to demonstrate the efficiency of interpretaton, not much emphasis was put on efficiency of representation.

Another directly executable intermediate language, DEL-Mary, does not have any specific formats. Operation codes are followed by one, two or three addresses and instruction sizes are multiples of bytes. The length of the instruction is determined by the instruction itself. A 65% reduction in program size and 32% reducton in the number of instructions have been claimed. This could possily be due to the type of storage (main and registers) used.

Tanenbaum's EM-1 [TANE78] instructions require an operation code and an index. The instruction set uses four formats, which are independent of probability of the instructions occurrence, and very similar to those of the Burrough's B1700's. The instructions most frequently used have their operation code and index in only one byte, and the others use two, two or three, or four bytes depending on their frequency of use.

The first study of formats and their use in an intermediate language was in DELTRAN [HOEV74a, FLYN77]. The machine on which DELTRAN was implemented had both registers and stacks. For this type of memory the use of IBM 360/370 formats would increase the number of instructions by at least twofold. The format type is best specified separately. DELTRAN requires specifying each operand only once in any instruction. For example instructions of the form X = A + A should have only one A in their representation (such as + A).

Specifying formats within the instructions is not suitable for directly interpretable representations. Specifying them separately is necessary. However, in a canonical interpretive form [FLYN77] the format which indicates the type, order and source of operands is a space overhead. This overhead can be reduced.

The three different kinds of operations commonly used are:

1. nullary, which does not have any operands,
2. unary, which has one source operand and produces one result, and
3. binary, which takes two source operands and produces a single result.

The formats for these three kinds of transformations when used for a registers + stack type of architecture are best stated separately. Encoding formats with the instructions would increase the number of instructions enormously.

Seven denotations are sufficient to represent possible sources or destination of operands. They are as follows:
just above the top of the stack (S''), top of stack (S'), just below the top of stack (S), A memory location (or register A, B, C) and a 'null' operand(__).

With these types of operands, the number of possible combinations for operations with up to three operands is 7^3 or 343. But for reasons of practicality most of them are illegal, redundant or too difficult for the compiler to generate. For example instructions such as S'SS'', S''S__. are illegal with respect to stack properties. Also, the

permutations of the three storage addresses are redundant.

It has been shown [HOEV77] that after removing the illegal and redundant combinations:

1. Thirty–two formats are sufficient to express all legal transformations using up to three operands.
2. For those using only one or two operands only sixteen formats will be enough.
3. With symmetry in use of operands only eight formats are sufficient.
4. Six formats are sufficient if the unique identifier property of canonical interpretive form [FLYN77] is to be ignored.

The method of choosing the right number of formats depends on the needs of the programs. For a complete set thirty–two formats are required. These can be represented by five bits. Even if the cannonical interpretive form can be achieved these five bits would be an overhead for every instruction.

One method of saving on the total space required for the storage of formats is to use techniques that allocate different lengths depending on their probabilities. This method introduces some complexity in compiling. We will discuss two other methods.

METHOD 1

One method of specifying different formats is to classify them on the basis of the number of operands instead of the types of sources and destinations. The four classifications are:

1. no operands, for operation codes such as continue, stop and end.
2. one operand which is also the destinaton of the result. For instructions such as A = A + A or $B = B^2$
3. two operands in which case there are two operands and three possible combinations.
   a. operate on the first and store in second. eg. B = A + A.
   b. operate on both and store in first. eg. A = A + B.
   c. operate on both and store result in second. eg. B = A + B.
4. Three operands: In this case there are three operands and two possible combinations.

a. operate on first two in order and store in the third

b. operate on first two in opposite order and store in third

These four different types require seven different representations. These seven can be shown to be functionally complete for operations requiring upto three operands

op(a)

a <-- op(a)

b <-- op(a)

a <-- op(a, b)

b <-- op(a, b)

c <-- op(a, b)

c <-- op(b,a)

an <-- op(a1,a2,..an)

The representation of the first seven types requires three bits. The last representation (three bits give eight) can be used for operations which has many operands with the same operation, for example A = A1 + A2 +..+ An. This type of operation is not very common but still could be used for further compaction. This set of format representation gives a further saving of 40% in the space required for formats. If stack addresses are specified as zero a maximum of five bits will be required in the worst case. The other advantage of this method is the flexiblity in having a large number of operands when the opcode is the same. For example A = Q + W + R + T + Y + U can be represented with only one operation code.

Method 2

In this method four different types are specified by two bits and two of them, the third and fourth in method 1, need one or two more bits to give the complete information.

00 no operands (type 1)

01 one operand in which both the source and destination are the same

(type 2).

10 two operands, with a possibility of having a

a) third bit specifying address of result, or

b) next two bits giving order and destination. (type 3)

11 three operands and two different orders that can be specified by one bit. (type 4)

This method requires only two bits for symmetric representations as opposed to three in the technique used for DELTRAN. Its extension to specify the order of operation and destination of result requires an increase in the size of types 3 and 4 only. Assuming that all formats are used with the same probability this method, with the extensions, would require an average of 2.5 bits.

One advantage with both the methods is the possibility of having an unused representation for an opening to give bigger patterns for other uses (similar to that of B1700's 4-6-10).

3.4 Techniques for reducing execution time and space

Various techniques for improving the execution time and space for program's representation are being developed. They are to some extent independent. Decrease in size may increase the speed by reducing the fetch time but is usually offset by the decoding complexity. For this reason methods designed are difficult to argue for as claimed. Space reductions from 5% to 75% have been claimed. But the overheads are not very well known.

Initial work on directly interpretable representations was done to obtain an interpretable representation (such as high level language Euler's reverse polish form). The representation itself being much more efficient than conventional methods, not much emphasis has been placed on techniques for further program compaction to save time and space still more.

There have been many arguments advanced regarding the importance of space over time [SAUN79]. Some of them are as follows:

1. Space complexity is more important for real applications. A user can afford to wait a little longer with a complexly encoded program than with a very lengthy one

which does not fit into the storage.

2. Compact programs have other advantages such as less page faults in a paging environment, a small working set etc., which all indirectly save time.
3. When the dominating factor in execution time is the fetch time, a compact representation would be advantageous.
4. Since space compactions are static it is easier to measure achievement.

There have been many approaches to increasing speed by using techniques such as defining new instructions.

A few other techniques to gain program speed are by Abdallah-Kaalagrad [ABDA74], Rauscher [RAUS76], and Burrough's SD-I's [WILN72].

Apart from this, many statistical studies have been made to help design machines and instructions, but these are rather static analyses. Most of the techniques for code compaction were developed after studying a certain class of programs or programs written in a certain environment. The machines, compilers and customizers so designed cannot be claimed to be general. The efficiencies claimed will not hold in general and an ultimate goal would be to develop methods that treat individual program or a class of programs independently. The latest work done in this area is by Saunders [SAUN79] and Hehner [HEHN76].

3.4.1 Code Compaction

Hehner's Work on Code Compaction

Hehner devised techniques to make the machine representation of programs compact. His technique can be applied to a single program or a class of programs. It is an iterative method in which frequently used pairs of instructions are selected and replaced by one instruction and the iteration repeated until no more pairing is possible. Then, depending on the frequency of the resulting instructions, instruction length is determined. The technique is efficient to improve instruction sets and is probability dependent. In the tests performed, the technique increased the set of operators from 47 to 178 operation codes, decreased the space required for operations to less than half the space required

before pairing and to less than a quarter of the 8 bit fixed-length encoding.

Another method devised by Foster and Gonter [FOST71] is also probability dependent and works as follows:

For any operation code every operation has a known probablity of following it. This property is used to give minimum space encoding in the context following that operation. This is a coding technique and results of its implementation by Hehner on his sample gave a space reduction of about 50%.

Hehner's methods are efficient to produce compact representation and invent new instructions suitable for the program. The results have also been very promising, saving upto 75% on machine representation space. Though Hehner did this for machine language representation the techniques can be easily implemented for DIRs.

Saunder's Work On Code Compaction

Saunders work can be regarded as an approach towards ultimate goal in program representation. Earlier work was to develop better representations of a language. Then by collecting statistics and taking the most frequently used operands etc., appropriate compilers are designed to suit the environment from which the data was collected. These two methods have achieved some success but are too general. Saunder's idea which is not very different from Hehner's, treats each program independently and develops a compact representation for the same. Compaction is done on the basis of the frequency of instructions in that particular program and not some general statistics from a particular environment.

His approach is to customize a DIR. He admits the difficulty in measuring the efficiency and restricts tailoring to suit 'a translator, a DEL, and an interpreter which together implement some source language on some host machine.

DIR's are tailored to optimize the translation time, simplicity in translation, the resulting DIR's transperancy, and the interpreter's size and efficiency. Tailoring requires many compromises and tradeoffs. The goal and achievements of tailoring depend on the measuring parameters.

The choice of source language and the host chosen constrain a DIR designer. Some repeated sequences of certain operations in a language (such as array addition in

FORTRAN) or a sequence of instructions to implement an instruction (on the machine) frequently used by the user are among the constraints.

Apart from the constraints listed above, other organisational representations such as data references, code references and control information can be made compact by taking into consideration the statistical properties such as probabilities of occurrences in the program.

After a DIR is tailored an individual program can be customized to give the most suitable and compact representation. There are many ways of doing this. From defining opcodes ideal for the program to combining a few opcodes to create new opcodes. The latter is much cheaper to implement.

Interpreters can be customized by letting the user not pay for the facilities he's not using. If a certain operation is not used then remove it from the interpreter (by removing the link). This reduces the size of the interpreter too.

DIR's can be customized by replacing similiar sequences of instructions by a single instruction or call to a macro. This is similiar to Rau's P-DER. Saunder's method is to translate the program to a tree form and detect matching patterns. The matching patterns found are implemented as an instruction and their occurence replaced by the new instructions and proper parameters.

The techniques developed were used to built a compiler XP^4 for Pascal which customizes DIR's by omitting unused operations from the interpreter, adding new opcodes. An average compaction of about 50% on implementing the above, for a compiler for PASCAL written in LISP, has been claimed.

Conclusions

In this thesis we have examined the problem of matching program representations on existing architectures, and the problem of evaluating architectures. In Chapter 2 analytical arguments for interpreting and conditions for executing with writable control stores are surveyed, followed by the derivation of conditions under which interpreting would be efficient.

In Chapter Three we looked into the needs for and characteristics of high level DIRs. Different types of instruction formats, were surveyed and compared. Then techniques to further compact high level DIRs were discussed.

The results can be summed up as follows:

Current architectures are not suitable for a good mapping of high level language onto machine. Restricting the machine to a single high level language is also not ideal because different languages are good for different types of problems. A general architecture for all problem domains is very difficult to design and has not proved optimal. The best approach is to reduce languages to an intermediate form and emulate a soft architecture on the host machine.

This approach has been made in the Burrough's B1700, but it had different intermediate forms for each high level language, which required loading the appropiate interpreter for each intermediate form. A common intermediate form for a class of languages would be ideal. It is possible to translate a class of most commonly used high level languages to a common intermediate form. FORTRAN and PASCAL have already been translated to a common intermediate form.

DELTRAN as designed is not in a very compact form. It still can be improved. It is, however a good form to allocate resources and detect parallelism for further efficiency.

Building a common high level DIR is similiar to going back to the UNCOL, but with a different perspective. The goal is not portablity but direct execution. Efficient hosts for direct forms can be designed and implemented and facilities developed to debug the high level DIRs.

3.5 Suggestions for Future Research

Implementing DIRs has a long way to go. Translators for different languages to a common DIR need to be designed. Techniques to detect parallelism are required. It has been shown that more than half of execution time is spent on less than 4% of FORTRAN programs [KNUT74]. It should be possible to dynamically detect this and place the microcode for the 4% into control store and save on execution time.

Compact representations require bit machines. But the overhead for bit addressing is high. Hence instructions and formats should be such that they are suitable for byte/word machines. Interpreters for such machines need be developed.

In a multiprogramming environment it should be possible to load the appropiate interpreter for the langauge being interpreted. This can be accomplished by looking for commonality in interpreters and loading necessary parts as required.

Finally, most of the techniques discussed in this thesis apply only to static programs. Methods to detect problems of representations dynamically are needed.

References

[ABDA74] Abd-Alla A.M., and Karlgaard, D. C., *Heuristic Synthesis of Microprogrammed Computer Architecture.* IEEE Transactions on Computers, Vol. C-23, No. 8, August 1974. (pp 802-807)

[ABRM70] ABRAMS, P. S., *An APL Machine.* Ph. D. Dissertation, CS Dept., Stanford University, June 1970.

[ADAM79] Adams, W. S., *Comparison of Intermediate Language Machines.* Univ. Of Alberta Tech. Rep. No. TR 79-11, August 79.

[ANDE61] Anderson, J. P., *A Computer for Direct Execution of Algorithmic Languages.* Proceedings of the Eastern Joint Computer Conference, 1961.

[BATT78] Battarel, G. J., Chevance, R. J., *Design Of A High Level Language Machine.* C. I. I.,-Honeywell-Bull, Louveciennes - France, Computer Architecture News, Vol. 6 No. 9 June 1978.

[BAUR68] Bauer, H. R., Becker, S.I. and Graham, S. L., *AlgolW Implementation.* Technical Report CS-98, CS dept., Stanford University, March 1968.

[BELL71] Bell, C. G. and Newell, A., *Computer Sructures : Readings and Examples.* McGraw-Hill Book Company, 1971.

[BRID80] Bridges, C. W., and Abd-Elfattah, M. A., *Direct Execution of C-String Compiler Texts.* Microprogramming Conference, 1980.

[BSIL75] Basili, V. R., and Tunner, A. J., *SIMPL-T: A Structured Programming Language.* U Of Maryland Computer Centre, Computer Note CN-14.2 August 1975

[CAPO72] Capon, P. C. et. al., *The MU5 Compiler Target Language and Autocode.* The Computer Journal, Vol. 15, pp 109-112.

[CHUY69] Chu, Y. and Cannon, E.R., *High Level Language Memory Structure.* U Of Maryland, Computer Science Centre, College Park MD 20742 (TR 409)

[CHUY75] Chu, Y., (editor) *High Level Language Computer Architecture,* Academic Press NY 1975.

[COOP80] Cooper, R. E. M., *The Direct Execution Of Intermediate Languages On an Eclipse Computer.* SIGmicro newsletter, March 1980, Vol 11 No. 1.

[DAHL68] Dahl, O., Myhrhaug B., and Nygaard K., *SIMULA 67 Common Base Language,* **Publication No. S-2, Norwegian Computer Center, Oslo 1968.**

[DENN79] Dennis, J. B., Fuller, S. H., Ackerman, W. B., Swan, R. J. and Weng, K. *S.Research Directions in Computer Architecture,* in **Research Directions in Software Technology, Ed. P. Wegner, The MIT Press 1979, pp 515.**

[ELSW79] Elsworth, E. F., *Compilation Via An Intermediate Language.* **The Computer Journal, Sept 1979.**

[FEUS73] Feustal, E. A., *On the Advantages of Tagged Architecture.* **IEEE Transactions on Computers, Vol. C-22, No. 7, July 1973, pp 644-656.**

[FLIN77] Flink II, C. W., *EASY: The Design And Implementation Of An Intermediate Language Machine.* **Tech. Rep. NSWC/DL TR 3765 Dahlgren Virginia December 1977 (Naval Surface Weapons Centre, Dahlgren Lab.).**

[FLYN71] Flynn, M. J., and Tucker, A. B., *Dynamic Microprogramming: Processor Organization and Programming,* **CACM Vol 14, No. 14, April 1971.**

[FLYN74] Flynn, M. J., *Trends And Problems In Computer Organizations.,* **IFIP proceedings 74, North-Holland Pub., (pp 3-10).**

[FLYN77] FLYNN, M. J., *The Interpretive Interface: Resources And Program Representation In Computer Organization.* **High Speed Computer And Algorithm Organization (pp 41-69), Academic Press., Inc.**

[FLYN78] Flynn, M. J., *A Cannonic Interpretive Program Form For Measuring "IDEAL" HLL Architecture.* **Computer Architecture News, Vol. 6 No. 8 April 1978.**

[FLYN80] Flynn, M. J., *Directions and Issues in Architecture and Languages.* **IEEE Computer, October 1980 (pp 5-22).**

[FLYN80a] Flynn, M. J. and Hennessey, J. L., *Parallelism and Representation Problems in Distributed Systems.* **IEEE Transactions On Computers, Vol. C-29, No. 12, December 1980 (pp 1080-1086).**

[FOST71] Foster, C. and Gonter A. *Measures of Opcode Utilization.* **IEEE Transactions on Computers, Vol. C-20, January 1971, (pp 108-111).**

[FULL77] Fuller, S. H., Shaman, P. S., and Lamb, D. A., *Computer Architecture Selection Committee Final Report, Volume III- Evaluation of Computer Architecture Via Test Programs.* **U. S. Army Technical Report ECOM-4528, September 1977.**

[HAMM72] Hammerstrom, D. W., and Davidson, E. S., *Information Content of CPU*

Memory Referencing Behavior. IEEE Transactions on Computers, Vol. C-21, No. 9, pp 948-960, September 1972.

[HAYN73] Haynes Leonard S. *The Architecture of an Algol 60 Computer implemented with distributed processors.* Naval Surface Weapons Center, White Oak, Silver Spring, Maryland 20910

[HAYN73a] Haynes, Leonard S. *The structure of an AIGOL-60 Polish String Language.* ACM IEEE Symposium on High Level Language Computer Arch (Proceedings) U of Maryland, College Park, Maryland, November 1973, pp 131-140.

[HEHN76] Hehner, E. C. R., *Computer Design To Minimize Memory Requirements.* Computer, August 1976 (pp65-70).

[HEHN76a] Hehner, E. C. R., *Information Content Of Programs And Operation Encoding.* JACM Vol. 24, No. 2, April 1977, pp 290-297.

[HOEV74] Hoevel, Lee W., *"IDEAL" directly executable language: an analytical argument for emulation.* IEEE Transactions on Computers, Vol C-23 No. 8, Aug 74, pp 759.

[HOEV74a] Hoevel Lee W. *Languages for Direct Execution.* SigMicro7, pp 307-316, 1974.

[HOEV75] Hoevel, L. W. and Wallach, A. W., *A Tale of Three Emulators.* Tech. Rep. No. 98, Digital Systems Laboratory, Stanford Laboratory, California. November 1975.

[HOEV77] Hoevel, L. W., and Flynn, M. J., *The Structure of Directly Executed Languages: A New Theory of Interpretive System Design.* Tech. Rep. No. 130., Digital Systems Laboratory, Stanford University, March 1977.

[HOEV78] Hoevel, L. W., *Directly Executed Languages.* Ph. D. Thesis, The Johns Hopkins University, 1978.

[HOEV79] Hoevel, L. W., and Flynn, M. J., *A Theory of Interpretive Architectures: Ideal Language Machines.* Tech. Rep. No. 170, Computer Systems Laboratory, Stanford University, February 1979.

[KNUT70] Knuth, D. *An Empirical Study of Programming Languages.* Computer Science Department, Tech. Rep. CS 186, Stanford Univ., 1970.

[KNUT71] Knuth, D. E. *An Empirical study of FORTRAN programs.* Software Practice and Experience Vol 1 No. 2, April 1971.

[LAWS68] Lawson, H. W. Jr. *Programming-Language-Oriented Instruction Streams,* IEEE Transactions on Computers. Vol. C-17, No. 5, May 1974, (pp 476-485).

[LAWS71] Lawson, H. W. and Burton, K. S. *Functional Characteristics Of a Multilingual Processor.* **IEEE Transactions On Computers, Vol. C-20, NO. 7, July 1971.**

[LIND72] Lindsey, C. H., *Making the Computer Fit the Language.* **in Algol68 implementation, J. E. L. Peck ed. 1972.**

[LIUP78] Liu, P.S. and Mowle F. J. *Techniques of Program Execution with a Writable Control Memory.* **IEEE Transaction on Computers, Vol. C-27, No. 9, September 1978, (pp 816-827).**

[LUND77] Lunde, A., *Empirical Evaluation of Some Features of Instruction Set Processor Architectures,* **CACM Vol 20, No. 3, March 1977.**

[MCKE67] McKeeman, W. M. *Language Directed Computer Design* **Proccedings of the Fall Joint Computer Conference, AFIPS Press, 1967.**

[MELB65] Melbourne, A. J. and Pugmire, J. M., *A Small Computer for the Direct Processing of FORTRAN Statements.* **The Computer Journal, Vol. 8, pp 24-27.**

[METR80] Metropolis, N., Howlett, J. and Rota, G. (editors), *A History of Computing in the Twentieth Century.* **Academic Press 1980.**

[MULL63] Mullery, A. P., *ADAM - A Problem Oriented Symbol Processor.* **Proc. of the SJCC, pp 367-380.**

[MYER78] Myers, G. J., *Advances in Computer Architecture.* **John Wiley & Sons, 1978.**

[MYER78a] Myers, G. J., *The Evaluation of Expressions in a storage-to-storage Architecture.* **Computer Architecture News, Vol. 6 No. 9 June 1978.**

[NEUH80] Neuhauser, C. J. *An Emulation Based Analysis of Computer Architecture.* **Ph. D. Thesis, The Johns Hopkins University, 1980.**

[ORGA78] Organick, Elliot I., and James, Hinds A., *Interpreting Machines: Architecture and Programming of the B1700/B1800 Series.* **1978 North-Holland New York.**

[RAUR78] Rao, B. R., *Levels of Representation of Programs and the Architecture of Universal Host Machines.* **Sigmicro Newsletter Vol. 9, No. 4; December 1978.**

[RAUS76] Rauscher T. G. and Agarwala, A. K., *Developing Application Oriented Computer Architectures on General Purpose Microprogrammable Machines.* **National Computer Conference, 1976, pp 715-722.**

[RICE71] Rice, R and Smith, W. R., *SYMBOL - A major Departure from Classic*

Software Dominated von Neumann Computing Systems. **Proc. SJCC, 1971 pp 575-587.**

[RICH71] Richards, M., *The Portability of the BCPL Compiler.* **Software Practice and Experience, Vol 1 No. 2, April 1971.**

[SAUN79] Saunders, S. E. *Compiling Customized Executable Representations and Interpreters.* **Carnegie-Mellon University Tech. Rep. CMU-CS-79-127, June 1979.**

[STEV79] Stevenson, J. W. and Tanenbaum, A. S. *Efficient Encoding of Machine Instructions,* **Computer Architecture News, Vol. 7 No. 8 June 1979.**

[STRG58] Strong, J. et al., *Report on the Shared Ad-hoc Committee on Universal Languages - A Proposed Solution,* **CACM 1958**

[SUGI69] Sugimoto, M. *PL/1 Reducer and Direct Processor,* **Proc. ACM Conf. 1969, pp. 519-538.**

[TAFV75] Tafvelin, S. and Wikstrom, A. *Aspects Of Compact Programs And Directly Executed Languages,* **BIT 15, 1975 (pp. 203-214).**

[TANE78] Tanenbaum, A. S. *Implications of Structured Programming for Machine Architecture,* **CACM Vol 21 No 3, March 1978, (pp 237-246).**

[VAND62] Van der Poel, W. L. *The Construction of an Algol Translator for a Small Computer, in Symbolic Languages in Data Processing,* **Gordon and Beach, New York, 1962.**

[VICT73] Basili, Victor R., *SIMPL-X: Univ. of Maryland, Computer Science Centre, TR-223 Jan. 1973.*

[WEBE67] Weber, H., *A Microprogrammed Implementation of EULER on IBM System/360 Model 30.* **CACM, Vol. 10, No. 9, September 1967, pp 549-558.**

[WICH69] Wichman, B. A., *A Comparision of Algol 60 Execution speeds.* **National Physical Laboratory Report CCU-3, January 1969.**

[WILN72] Wilner, W. T., *Design of the Burroughs B1700,* **FJCC, 1972 (pp 489-497)**

[WILN72] Wilner, W. T. *Burroughs B1700 memory utilization,* **FJCC 1972 (pp 579-586).**

[WIRT65] Wirth, N. and Weber, H., *EULER: A generalisation of ALGOL and its Formal Defination,* **Stanford Univ. Tech. Rep. 1965, STAN-CS-65-20 (PB 176755)**

[WORT73] Wortman, D. *A Study of Language Directed Compiler Design* Ph. D. Dessertation, Stanford Univbersity, 1973, Stanford University, Calif., 1973.

[ZAKS71] Zaks, R., Steingart, D., and Moore, J., *A Firmware APL time-sharing System.* Proc. AFIPS SJCC 38, pp 179-190.

B30322